DTIC FILE COPY A227 675 ②

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 3451-4-14/2	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION SofTech, Inc. / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Aeronautical Systems Division SCOL
6c. ADDRESS (City, State, and ZIP Code) 3100 Presidential Drive Dayton, OH	7b. ADDRESS (City, State, and ZIP Code) Building 676 Area B Wright Patterson AFB, OH 45433
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U.S. Army, CECOM / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33600-87-D-0337
8c. ADDRESS (City, State, and ZIP Code) Ft. Monmouth, N.J. 07703-5001	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

DTIC ELECTE SEP 27 1990 D E

11. TITLE (Include Security Classification)
General Architecture Study (Unclassified)

12. PERSONAL AUTHOR(S)
Quanrud, Richard B.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 9/87 TO 01/88	1988 January 22	104

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Reusable Software Command and Control Software Ada

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Generic architectures provide an approach to the development of reusable software for families of related applications. They provide both a high level design and a set of reusable components to be used in the applications supported by the architecture. The components are typically larger and more complex and result in higher levels of software reuse than with conventional reusable components. The study explores the use of the Ada language in the development of generic architectures for Army command and control applications.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Gerald Brown	22b. TELEPHONE (Include Area Code) (201)544-2685	22c. OFFICE SYMBOL AMSEL-SI-SD-S-H

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: - **GENERIC ARCHITECTURE STUDY**

Final Report

CIN: **C04-038NN-0001-00**

22 JAN 1988

GENERIC ARCHITECTURE STUDY:

Final Report

PREPARED FOR:

U S ARMY CECOM
CENTER FOR SOFTWARE ENGINEERING
AMSEL-RD-SE-AST
FORT MONMOUTH, NJ 07703

PREPARED BY: SofTech Inc.
460 Totten Pond Road
Waltham, MA 02254

22 JAN 1988

Final Report, January 22, 1988
Interim Report, November 1987

SofTech, Inc.

460 Totten Pond Road
Waltham, MA 02254

Contents

PREFACE TO THE FINAL REPORT vi

EXECUTIVE SUMMARY vii

1 GENERIC ARCHITECTURES 1

1.1 Introduction 1

1.1.1 Concept 1

1.1.2 Origins 3

1.1.3 Relationship to Reusable Component Libraries 3

1.1.4 Advantages and Disadvantages 5

1.2 Reusable Component Requirements 7

1.2.1 Encapsulation and Abstraction 7

1.2.2 Adaptation 8

1.2.3 Support for Object-Oriented Design 9

1.3 Ada Language Support 10

1.3.1 Encapsulation and Abstraction 11

1.3.2 Adaptation 13

1.3.3 Object-Oriented Design 19

1.4 Implementation of Generic Architectures 21

1.4.1 Domain Definition 22

1.4.2 Design 23

1.4.3 Component Implementation 24

1.5 Summary 25

2 ARMY COMMAND AND CONTROL REQUIREMENTS 27

2.1 Introduction 27

2.1.1 Purpose and Scope 27

2.1.2 Major Influences 28

2.2 Functional Requirements 31

2.2.1 System Management 31

2.2.2 Data Base 31

2.2.3 Information Processing 32

2.2.4 Man-Machine Interface 34

2.2.5 Decision Support 35

2.2.6 Communications Processing 37

2.3 Other Requirements 38

2.3.1 Security 38

2.3.2 Training 38

2.3.3 Performance 39

2.3.4 Flexibility 39

2.3.5 Reliability 39

2.3.6 Interoperability 40

2.3.7 Maintainability 40

2.3.8 Portability 40

2.3.9 Efficiency 41

2.4 Reusability Analysis 41

2.4.1 Dimensions of Reusability 41

2.4.2 Analysis by Function 43

2.4.3 Data Driven Operations 45

2.5 Summary 46

3 A STRAWMAN GENERIC ARCHITECTURE FOR COMMAND AND CONTROL 48

3.1 Purpose of the Strawman 48

3.2 The ACCS Environment 49

3.2.1 Hardware 49

3.2.2 Software 50

3.2.3 Man-Machine Interface 51

3.3 The Design 51

3.3.1 Flow of Control 52

3.3.2 Objects and Components 55

3.4 The Processing of Outgoing Messages 58

3.4.1 Principal Requirements for Outgoing Messages 58

3.4.2 Operations on Outgoing Messages 60

3.4.3 Component Dependencies 63

3.5 Application of the Concepts of Generic Architectures 63

3.5.1 Components Represent Abstract Objects 63

3.5.2 Encapsulation of Data and Operations 65

3.5.3 Integrated Nature of the Components 66

3.5.4 Size and Complexity of the Components 67

3.5.5 Adaptation of the Components 68

3.5.6 Requirements for Additional Components 70

3.6 Summary 71

A Appendix: BIBLIOGRAPHY **72**

B Appendix: DEMONSTRATION AND VALIDATION **74**

B.1 Introduction 74

B.1.1 Role of the Sponsor 75

B.1.2 Tasks 75

B.2 Domain Analysis 76

B.2.1 Purpose 76

B.2.2 Problem Domain vs. Architectural Domain 77

B.2.3 The Domain Analysis Process 77

B.3 Development of the Prototype Generic Architecture 84

B.3.1 Purpose 84

B.3.2 The Development Process 85

B.4 Development of The Prototype Applications 87

B.4.1 Purpose 87

B.4.2 The Development Process 88

B.5 Summary 89

List of Figures

1 Encapsulation of a reusable component in an Ada package 12

2 Encapsulation of a reusable component in an Ada generic package 14

3 Encapsulation of application specific code in an Ada package . . . 15

4 Instantiation of an Ada generic package to produce a reusable component 15

5 The declaration of a separate subunit in a package body 17

6 The implementation of a separate subunit 17

7 Use of overloading to provide alternative application specific components 18

8 Selection of alternative application specific components through generic instantiation and overloading 19

9 ACCS functional segments and communications support 29

10 Main event loop for the operator task 53

11 Menu/Function key command processing 54

12 Component dependencies in the processing of outgoing messages . 64

List of Tables

I Likelihood of Component Reuse Across ACCS Functions 44

II Operations performed on messages by the message manager component 61

III Operations performed on other components that are invoked by operations of the message manager component 62

PREFACE TO THE FINAL REPORT

This report explores the use of generic architectures as an approach to the development of reusable software in Ada. It focuses specifically on their applicability to the development of Army command and control systems. In that respect it is a further extension of work performed by SofTech on generic architectures for the RAPID[1] program of the Army Information Systems Engineering Command and for the RAPIER[2] system of the Air Force Space Command. The report also gives special attention to the use of the Ada programming language in the development of generic architectures.

The report examines the concepts of a generic architecture as well as the requirements of Army command and control systems. It then illustrates the application of the concepts through a "strawman" generic architecture for command and control. The final chapter discusses the activities and issues of importance in the demonstration and validation of this approach to reusable software. (KR)

The study that produced this report has been sponsored by the Software Engineering Division of the U.S. Army Communications-Electronics Command (CECOM), Ft. Monmouth, N.J. and conducted under SofTech's Language Control Facility Contract[3] with the U.S. Air Force Aeronautical Systems Division, Wright-Patterson AFB, Ohio.

[1] Reusable Ada Packages for Information System Development

[2] Rapid Emergency Reconstitution System

[3] Contract number F33600-87-D-0337.

EXECUTIVE SUMMARY

Generic architectures provide an approach to the development of reusable software for families of related applications. Interest in generic architectures is based on their potential for achieving significantly higher levels of software reuse than with traditional libraries of reusable components.

A generic architecture provides a high level design for a family of related applications and a set of reusable components that are intended for use in those applications. Orientation to a design and to the requirements of a family of applications allows the components to be larger, more complex, and more highly integrated than traditional reusable components. These are the features which can result in higher levels of software reuse in the applications within the domain of the architecture. These same features make a generic architecture less usable outside of the application domain for which it was designed.

Development of a generic architecture is appropriate only if the domain is large enough to justify the development costs. Different generic architectures may be required to support different application domains.

The implementation of an actual application that uses a generic architecture will require additional components that are not provided by the architecture. It will also require the adaptation of many of the architecture's reusable components. The report examines the ability of the Ada language to support the adaptation of those components in ways that do not result in permanent changes to the components themselves, i.e., in ways that do not affect their future reusability.

There are a number of important advantages in the use of a generic architecture. Because a substantial amount of software is reused in each application, both development and maintenance costs are reduced. It can have a significant impact on interoperability by ensuring the consistent application of message protocols and other standards that are implemented through the reusable components of the architecture. It can

also produce a degree of consistency in the man-machine interface that makes it easier to reassign operators to different applications across the domain of the architecture.

The principal disadvantages are the initial cost and the limitations on the use of the architecture outside of its intended domain. Additional effort is required to develop a generic architecture in order to consider the requirements of all of the applications within its domain and to build in the flexibility needed to adapt to the current and future requirements of the applications. Almost all of this expense must be incurred before a generic architecture can be used to support its first application.

This report gives considerable attention to the features of the Ada language that support the development of the reusable components for a generic architecture. It also discusses the role of object-oriented design techniques in the development process and the ability of Ada to fully support that technique.

Care must be taken in the selection of a domain for a generic architecture. It must have enough applications to justify the cost of development. The applications and operating environment must be sufficiently similar to permit the reuse of the overall design and a substantial number of components across the applications. Finally, the effort is most likely to be successful if there is a single organization, with responsibility for the applications, that can sponsor and support the effort.

Chapter 2 of the report examines ACCS, the Army Command and Control System, as a possible domain for a generic architecture. Based on an analysis of the system specification for the Maneuver Control System, there appear to be a substantial number of functions that could be implemented with reusable software across the the battlefield segments of ACCS. An analysis of the ACCS Functional Description indicates that a great deal of flexibility is needed in the system to handle the complex information flows in a task force organization. This requirement also tends to promote the use of software that could be reused across ACCS.

Chapter 3 outlines a "strawman" generic architecture for command and control to illustrate a number of the concepts and characteristics of this approach. It highlights the ability to build components that are larger and more complex in order to achieve higher levels of software reuse.

1. GENERIC ARCHITECTURES

1.1 Introduction

Generic architectures provide an approach to reusable software for families of related applications. This approach is somewhat different from that of reusable component libraries and can lead to significantly higher levels of software reuse. This section defines the concept, traces its origins, compares it to libraries of reusable components, and discusses its advantages and disadvantages.

1.1.1 Concept

A generic architecture provides a high level design for a family of related applications and a *set of reusable components that are specifically intended for use in those applications*. The reusable components are designed to work together and should provide most of the code that would be included in a typical application. Actual applications are developed by adding application specific components and adapting the reusable components to meet the requirements of the application. Adaptation of a reusable component may take the form of modification, extension, use-as-is, or replacement.

The central feature of a generic architecture is that it provides both a design and an integrated set of reusable components. The design and the reusable components are intended for use only by the applications within the domain of the architecture. That domain may be small or large depending on the number of applications that are sufficiently related to share a common high level design and a substantial number of common functions. It is not inconceivable that a generic architecture might be used to support an application outside its intended domain, but it is not intended for that purpose and one might expect less satisfactory results.

The effectiveness of this approach depends to a large degree on the ability to concentrate on a specific set of requirements associated with the application domain. It is much easier and cheaper to develop a reusable component to meet a set of specific requirements than to develop a component for more general use. This helps to limit the effort necessary to develop a generic architecture. Development is appropriate only if the number of applications in the domain is large enough to justify that development effort.

The implementation of an actual application based on a generic architecture will normally require additional components that are not provided with the generic architecture. These must either be developed to meet the immediate requirement or obtained from some other source such as a library of reusable components.

Furthermore, the components provided by the architecture generally require some amount of adaptation to meet the specific requirements of the application. If they can't be used in their original form, they must be modified, extended, replaced, or deleted. Of course, this must be done without changing the source code of the original components or they cannot be considered to be reusable. The programming language used to implement the components must permit the non-intrusive adaptation of the reusable components provided by the generic architecture.

It follows that separate generic architectures may be developed for different application domains. It is even likely that some architectures may share at least some of their reusable components. Left unanswered for now are the questions:

- How are the bounds of a domain established?
- How is the design developed?
- How are the reusable components implemented, and
- How is an application implemented using the architecture?

These issues will be discussed later in this chapter.

1.1.2 Origins

There is nothing particularly new about the concept of a generic architecture. Similar ideas are found in operating systems, which must be adapted to specific user configurations, and in application generators, intended to meet the requirements of a particular applications domain. MacApp, an application framework for the Apple Macintosh, is intended to provide this type of support for the development of new applications for that environment.

Dijkstra, [4] in his original paper on "Structured Programming" discusses "incomplete programs" constructed from "programming pearls" or reusable modules. Parnas [5] described techniques for developing and using reusable modules in the development of programs belonging to "program families". More recent contributions to the concepts of abstraction and object oriented design have done much to make generic architectures practical.

Generic architectures are an extension of the generic facilities of the Ada language, with which they are used in the reusability approach described in this study. The term "architecture" highlights the role played by the high level design that provides the basis for the reusable component structure.

1.1.3 Relationship to Reusable Component Libraries

The distinguishing characteristic between generic architectures and libraries of reusable components is that the design provided by an architecture is used in the development of the applications which use the architecture. The reusable components are intended for use only within the context of that design. Components in a library normally must be usable with designs that have yet to be determined. The generic architec-

[4] E. Dijkstra, "Structured Programming", Software Engineering Techniques Report on a Conference Sponsored by the NATO Science Committee, p. 84, October 1969.

[5] D. L. Parnas, "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, p. 1, March 1976

ture defines how the reusable components interact with each other so that they can be designed to work together.

In many cases, libraries are confined to a single application domain or a library has an internal organization that separates the components by application domain. It is then possible to make some assumptions about the type of design in which the components will be used. In a sense this is a movement away from libraries of general purpose reusable components and towards the type of capability provided by a more application specific generic architecture.

In summary, the components provided with a generic architecture are designed to meet the requirements of both the domain and the high level design. This orientation:

- Allows the components to be tailored more precisely to their intended use and reduces their generality,
- Allows them to handle more complex functions because the requirements of those functions may be more precisely specified, and
- Increases the chances that a given component will be reused in the implementation of an application within the intended domain.

The domain for the reusable components of a generic architecture is likely to be significantly smaller than that for a library due to the additional constraint that all of the applications using the components of the generic architecture must also use the same high level design. Within the applications of the domain, most of the reusable components will be reused most of the time.

This has a major impact on the relative efficiency of the two approaches. Libraries frequently contain so many components that special tools are needed for the selection and retrieval of the components appropriate to a given application. With the generic architecture approach, one starts with the assumption that most of the components provided by a generic architecture will be used. Each component must then be reviewed to determine whether it will be used as is, modified, extended, or deleted for a given application. Thus the amount of reusable code needed to achieve a given level of reuse with a generic architecture may be an order of magnitude smaller than that required to achieve the same level of reuse with the components of a more general purpose library.

This also has an impact on the way in which the components are documented under the two approaches. Documentation of a library component is intended to assist in the selection process, often supporting a decision among several variations that implement the same function. Documentation of a generic architecture component focuses instead on the role played by the component within the architecture and the ways that the component may be adapted to the requirements of a specific application. The latter approach should be taken to the documentation of any application specific components as well, in order to enhance their adaptability and support their reuse in prototype applications.

However, to some degree the two approaches can be complementary. Generic architectures almost never provide all of the components needed to meet the requirements of a specific application. Additional components are needed for functions that are not used widely enough within the domain to be included in the architecture. These components can often be found in libraries, particularly if they support common mathematical or other operations that are relatively domain independent.

1.1.4 Advantages and Disadvantages

There are a number of advantages in the use of generic architectures. First, the time and effort required to develop new applications is reduced substantially. They provide good support for the early prototyping activities needed to refine the requirements for new applications. The high level design is largely complete and places have been provided for the insertion of any additional components required by the application. Detailed design is required only for new components and for any adaptations to be made to the reusable components. Only the new components and adaptations must be coded and unit tested. Integration testing of the reusable components has already taken place so that only the new components and adaptations remain to be integrated. Performance and other risks are substantially reduced through the reuse of both a proven design and a substantial amount of tested code.

The effort required to maintain the resulting applications is reduced as well. Problems fixed in a reusable component are fixed in all of the applications that use that component and are avoided altogether by any future users of the component. A common high level design makes it easier for the maintenance programmer to develop the understanding of the design that is necessary to identify and isolate problems. Changes required to adapt to subsequent modifications in the hardware and software environment are likely to be confined to reusable packages. This is because components, e.g., device drivers and window managers, that interface with the hardware and software environment are among the most likely to be reused by applications which share the same environment.

Finally, one of the most important advantages is consistency in the *implementation* and behavior of the applications that share the architecture. Interface standards implemented through reusable components are easily enforced across major systems. Critical expertise can be applied to reusable components with benefits to all users. Equipment operators soon recognize the common "look and feel" of applications with a common design and shared components. This has an important impact on both the cost of training operators to handle new applications and the degree to which operators may reassigned from one application to another.

However, as with all reusable components, the components of a generic architecture may require more memory and execute more slowly than components designed to meet the needs of a single application. This is offset to an unknown degree by the greater impact of any effort spent on optimizations made to reusable components.

The most important problem is the "up front" cost of the development of the generic architecture itself. Virtually the entire cost of the architecture must be absorbed before it can be used to support its first application. This cost is increased by the need to:

- Develop an understanding of the common requirements of the entire domain of the architecture,
- Produce a design and structure the components in a way that anticipates the adaptations that may occur in the future, and
- Produce documentation to support future development activities as well as the ongoing maintenance of its components.

1.2 Reusable Component Requirements

The reusable components used with a generic architecture must meet several requirements. They must encapsulate the operations and data of the component and allow the user to treat the component as an abstract entity that hides the implementation of the operations and the data. They must also be adaptable to the requirements of specific applications without sacrificing reusability. These are also the attributes of object-oriented design: an approach to the development of software components that is discussed at the end of this section.

1.2.1 Encapsulation and Abstraction

A reusable component is the abstract encapsulation of the algorithms and data necessary to perform a specified set of functions. Encapsulation means that the component is self-contained, i.e., that all of the processing and data are defined within the component or within a clearly defined interface to the outside world. Abstraction allows the user to invoke the function through that interface, providing the input variables required by the function and obtaining the results in a usable form. It allows the user to make use of the component to perform a desired function without having to understand the processing that is carried out within the component.

These are the properties of a reusable component that allow it to be reused with less effort than an attempt to lift source code from one program and use it in another. In the latter case, it would be necessary to understand the reused code, to identify any references to code or data outside the selected code and to change the code to adapt it to its new context. It is not necessary to understand the code of a reusable component as long as the user understands how to use it and is confident that it correctly performs the required function.

1.2.2 Adaptation

It is easiest to define a reusable component for a simple well defined function such as the computation of a mathematical result. In most cases at this level, the reusable component is a single subroutine that operates on data supplied through its interface. Unfortunately, little of the code in most applications can be efficiently encapsulated and reused at this level. More complex reusable components are required, which generally include a number of subroutines and a certain amount of shared data.

These components become increasingly application specific as they grow larger or are used at higher levels in the design of the program. Even if they are intended to be reusable, they must reflect some assumptions about the context in which they are to be used. In the absence of such assumptions, they would become too large and complex to be used efficiently. The interface would also become so cluttered with the additional information required by the complexity of the processing that most users would elect to simply write the code required for the application in a non-reusable form.

There are two general approaches to reducing the complexity of higher level reusable components. The first is to allow the component to become more application and design dependent, to limit the range of applications and designs in which it may be reused. The second is to provide mechanisms for adapting the component to the requirements of a specific application, without changing the code of the component. Both methods must be used to meet the requirements of a generic architecture.

Allowing a component to become more application and design dependent simply amounts to dropping from consideration the requirements of some uses during the design process. For example, the components of an operating system are designed to support a specific set of functions. Variability is generally allowed only in the management of the resources of the system, e.g., main memory, disk space, and peripherals, which vary with the user's hardware configuration. Such restrictions can greatly simplify both the interface and the internal logic of a reusable component.

Adaptation of a reusable component requires mechanisms for the modification, extension, and replacement of the component. These mechanisms can be provided in the programming language, e.g., the generic facilities of Ada, or through programming, e.g., by including a call (a hook!) to an application specific subroutine outside of the reusable component that fills in some step of the processing.

1.2.3 Support for Object-Oriented Design

The principles of object-oriented design meet the above requirements sufficiently to justify discussion at this point. Object-oriented design provides an approach to the design and implementation of software components that are almost always reusable. It also places certain requirements on the programming language used to implement the components.

Object-oriented design (development or programming) encapsulates an abstract object (e.g., a window on a graphics display). The encapsulation defines all of the data needed to operate on an instance of the object (e.g., a single window) along with the methods (e.g., Ada subprograms) that can perform operations (e.g., display, scroll, resize, hide) on the object. A reusable component can represent a single object or a class of objects (e.g., all of the windows that can be shown on the graphics display).

All of the data associated with an instance of an object is internal to the reusable component that represents the object and can be manipulated only through the methods identified in its external interface. Thus, object-oriented design meets the encapsulation and abstraction requirements described above. A number of languages are available which can be used to implement objects, including Ada, Smalltalk, and object-oriented versions of Pascal and C.

The important new concept in object-oriented design is inheritance, which provides an excellent mechanism for the adaptation of a reusable component. Inheritance allows a new class of objects (e.g., a more application specific class of windows) to be defined as a subclass of some existing class (e.g., the more general purpose windows). It then inherits all of the types of data and methods associated with its parent class. However, it can have additional types of data, not available to the parent class, that it needs for objects of the subclass. It can also have additional methods as well as replacements for methods of the parent class. Thus, inheritance provides a convenient mechanism for encapsulating the changes to a reusable component that are needed to adapt it for a particular application (or set of applications) without physically changing the original reusable component. Support for some form of inheritance is needed for the efficient implementation of the reusable components of generic architectures.

1.3 Ada Language Support

This section examines the features of the Ada language that support the development of reusable components for generic architectures. The discussion follows that of the previous section. It first presents the Ada mechanisms for encapsulation and abstraction, goes on to discuss mechanisms for the adaptation of reusable components to the specific requirements of an application, and concludes with some remarks on Ada's ability to support object-oriented design.

1.3.1 Encapsulation and Abstraction

The Ada package is the primary language construct for the support of encapsulation and abstraction. It separates the interface of a reusable component from its implementation and limits the user to references to the interface. An Ada package has two parts, a specification (the interface) and a body (the implementation). Program references from outside the package are limited to declarations contained in the specification and this restriction is enforced by the compiler. Furthermore, the language supports the use of private data types in the specification part so that outside references can be made to a type, but not to the details of the implementation of that type.

For example, consider the fragments of an Ada package contained in Figure 1.

Package A is a reusable component that encapsulates operations (subprograms) B and C. Both of these operate on data of type X. The specification part of the package makes the declarations of B, C, and X available for use (exports the declarations) outside the package. However, the variables Y and Z, as well as U, the data type Q, and procedure E, are not exported and may not be referenced outside of package A. This is because Y and Z are declared within the private part of the specification of package A, and U, Q, and E are declared within the body of the package. However, all of these may be referenced by the code inside the body of package A that implements subprograms B and C.

It is apparent from the limited information contained in the example that procedure B performs some type of operation on a record of type X. Perhaps it initializes a record or adds it to a list of similar records. Function C answers some question about a record of type X and returns a Boolean (true/false) result. In both cases, code outside the package may direct that the operation be executed, but it cannot affect the way that it is executed. That processing is completely encapsulated within package A. Likewise, code outside the package may declare data of type X, but it cannot operate on Y and Z, the internal components of X, except by executing subprograms B and C.

Package Specification:

```
package A is
  type X is private;
  procedure B (P:X);
  function C (P:X) return boolean;
private
  type X is record
    Y: boolean;
    Z: integer;
    end record;
end A;
```

Package Body:

```
package body A is
  type Q is record
    .
    .
    end record;
  U: Q;
  procedure E (V:Q) is
    begin
      .
      .
    end E;

  procedure B (P:X) is
    begin
      .
      .
    end B;
  function C (P:X) return boolean is
    begin
      .
      .
       E(U);
      .
    end C;
  end A;
```

FIGURE 1. Encapsulation of a reusable component in an Ada package

Thus, package A not only encapsulates X, B, and C, but encourages the user to treat them as abstractions. They are abstractions because their details are beyond the user's control. Of course, there is nothing to keep the user from reviewing the program source code that provides these details, but an Ada compiler will not let the user refer to these details in any code that is not part of package A.

1.3.2 Adaptation

The Ada language provides a number of mechanisms for the modification or replacement of a reusable component. The most important of these are generics, separate subunits, and overloading. Each of these is discussed and illustrated below. Mechanisms for the extension of a reusable component are virtually non-existent. That problem is discussed in the section on object-oriented design that follows.

Generics

Generics provide the principal Ada mechanism for modifying a reusable component, contained in an Ada package, to meet the requirements of a specific application. A generic package is essentially a general purpose template for the actual component that will be used in an application. It provides for the compilation time substitution of type, value, object, and subprogram parameters. The substitution is performed by "instantiating" the generic package, i.e., by identifying the generic package and giving the compiler the parameter values that are to be substituted.

Figure 2 shows how the Ada package contained in Figure 1 might look in generic form.

Note that data type Q and procedure E have been removed from the body of package A and that they are now identified as generic parameters at the beginning of the package specification. Separated from A in this fashion, they can be changed to meet the requirements of different applications without changing package A itself. For

Generic Package Specification:

```
generic
  type Q is private;
  with procedure E(V:Q);
package A is
  type X is private;
  procedure B (P:X);
  function C (P:X) return boolean;
private
  type X is record
    Y: boolean;
    Z: integer;
    end record;
end A;
```

Generic Package Body:

```
package body A is
  U: Q;
  procedure B (P:X) is
    begin
      .
      .
    end B;

  function C (P:X) return boolean is
    begin
      .
      .
      E(U);
      .
    end C;
  end A;
```

FIGURE 2. Encapsulation of a reusable component in an Ada generic package

```
package D is
  type R is private;
  procedure G (T:R);
private
  type R is record
    .
    .
    end record;
end D;

package body D is
  procedure G (T:R) is
    begin
      .
      .
    end G;
  end D;
```

FIGURE 3. **Encapsulation of application specific code in an Ada package**

```
with A;
with D; use D;
package F is new A(R, G);
```

FIGURE 4. **Instantiation of an Ada generic package to produce a reusable component**

example, a new package, D, might be defined as shown in Figure 3.

The data type R and the procedure G are declared in the specification of package D and are compatible in form with the generic parameters Q and E of package A. An actual component, F, for an application can be produced by instantiating the generic package A using the code shown in Figure 4. The third line of the figure indicates that the actual parameters R and G are to be substituted for the corresponding generic parameters, Q and E, of generic package A.

Package A can now be used in a number of different application contexts simply by providing other packages, similar to D, which satisfy its generic parameters. In like fashion, D could have been defined as a generic package. Thus, a larger subsystem or program can be constructed through a series of instantiations of generic components.

Separate Subunits

Ada also allows the implementation of a subprogram, package, or task to be treated as a "separate subunit" that is compiled separately from the reusable component in which it is declared. For example, the body of package A shown in Figure 1 might be recoded as shown in Figure 5. Note that the code that implements procedure E is no longer included in the body and is identified as "separate". The user of component A would then be required to supply a separate component, such as that shown in Figure 6, that contains the implementation of procedure E. The advantage is that the user is able to supply application specific details in procedure E that would otherwise reduce the reusability of component A.

The difference between the use of generics and the use of separate subunits is in the references that may be made by one component to declarations contained in the other. In the generic example above, package D must be compiled before package A can be instantiated to produce component (package) F. Therefore, the code in package A is allowed to make references to generic parameters that correspond to declarations contained in the specification of package D, but package D cannot make references to declarations contained in package A. With procedure E declared as a separate subunit, it must be compiled after package A and can make any reference to declarations contained in A that it would have been able to make if it had been included in A at the point of the "separate" declaration. Thus, in the components of an actual generic architecture, the direction of the references may determine the appropriate approach.

Overloading

Where it is known that there are a limited number of variations on an application dependent component, it is possible to use "overloading" in combination with generics

```
package body A is
  type Q is record
    .
    .
  end record;
  procedure E (V:Q) is separate;
  procedure B (P:X) is
    begin
      .
      .
    end B;
  function C (P:X) return boolean is
    begin
      .
      .
      E(U);
      .
    end C;
  end A;
```

FIGURE 5. The declaration of a separate subunit in a package body

```
separate (A)
procedure E (V:Q) is
  begin
    .
    .
  end E;
```

FIGURE 6. The implementation of a separate subunit

```
package D is
  type R is private;
  type S is private;
  procedure G (T:R);
  procedure G (T:S);
private
  type R is record
    .
    .
    end record;
  type S is record
    .
    .
    end record;
end D;

package body D is
  procedure G (T:R);
    begin
      .
      .
    end G;
  procedure G (T:S);
    begin
      .
      .
    end G;
  end D;
```

FIGURE 7. **Use of overloading to provide alternative application specific components**

to select the appropriate variation for each use. Overloading in Ada allows the same name to be used in different declarations. The compiler must then select the correct declaration based on the context in which the name is being used. Figure 7 provides a revised version of package D (from Figure 3) that illustrates this point. Package D now includes two versions of procedure G, one that has an argument of type R and the other of type S. The compiler will select the correct one on the basis of the data types R and S.

Figure 8 shows the instantiation of the generic package A (from Figure 2) for each data type and version of procedure G. The first version of procedure G will be used if A is instantiated with type R; the second version will be used if A is instantiated with type S. Package D is now a reusable component that provides the variations on operation E that might be required for different applications.

Instantiation for type R and procedure G(T:R):

```
with A;
with D;
package F is new A(D.R, D.G);
```

Instantiation for type S and procedure G(T:S):

```
with A;
with D;
package F is new A(D.S, D.G);
```

FIGURE 8. **Selection of alternative application specific components through generic instantiation and overloading**

1.3.3 Object-Oriented Design

Ada packages may be used to represent individual objects or classes of objects in the context of object-oriented design. A package represents a single object if the data variables used to represent the object are allocated statically within the package. A package represents a class of objects if the data variables for each instance of an object are included in a record which is allocated dynamically in memory. In such cases, the declaration of the record is exported under the restrictions of a private type so that other components can refer to a specific instance of an object without being able to directly manipulate the data associated with that instance. Only the subprograms of the package which represents the object or class of objects are able to perform such operations. Of course, an outside component may call these subprograms to execute operations on the data of an object.

There is some controversy about whether Ada is truly an object-oriented language.

Grady Booch[6], perhaps the most popular authority on the use of the language, insists that Ada is an object-oriented language and advocates the use of object-oriented development as a general technique for the implementation of software in Ada. The controversy surrounds the issue of inheritance, for which there is very limited support. Booch recognizes these limitations but questions the need for such support in object-oriented software developed in Ada.

True inheritance in Ada would allow the programmer to encapsulate all of the changes to a reusable component represented by an Ada package in a separate package so that there would be no change to the package of the original reusable component. The changes in the separate package would include additions, replacements, and deletions to the operations that may be performed on the object represented by the reusable component. They would also define additional variables for each instance of such an object.

The previous section discussed ways in which operations could be replaced as long as the changes were anticipated in the development of the original component. Deletion can be handled through the replacement of the subprogram for an operation by a null subprogram, one that doesn't do anything.

The real limitations are in the addition of new operations and variables. It is possible to add additional operations in a separate package if the original package exports the full declaration of the record type that contains the variables for an instance of a object. That record can then be redefined as a "derived type" in the separate package that contains the changes. Derived types in Ada inherit all of the operations that may be performed on objects of the original type. Operations may now be defined in the separate package that add to, delete, or replace operations contained in the original component.

The problem with this approach is that the declaration of the record type is no longer private. It is necessary to export the details on its constituent variables for

[6]G. Booch, "Object Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, p. 211, February 1986.

use in the separate package so that new operations can be defined which operate on those variables. However, in doing this, the same information is made available for use by any component in the application. An important element of abstraction has been lost and it is no longer possible to be certain that all operations on the object represented by a reusable component take place through the approved interface to that component.

Finally, there seems to be no easy way to add variables to those already defined for an object. Such variables might be needed to support a new operation or set of operations on an object. The only alternative seems to be to develop a new version of the original reusable component that contains additional variables. Now the original component is no longer reusable, at least not in the current application.

More experience is needed in the development of Ada components for a generic architecture to fully understand the importance of these limitations on inheritance. It is likely that inheritance is of greater practical importance in the development of components for a generic architecture than it is in the development of other components using object-oriented techniques. This is because of the greater need for non-intrusive techniques for modifying and extending the often complex and highly integrated components used to support a generic architecture. However, experience with the MacApp framework for the Apple Macintosh has shown that a large share of the components can be reused without modification. It is also possible that ways will be found to "program around" these limitations in actual practice.

1.4 Implementation of Generic Architectures

In many respects the implementation of a generic architecture is like that of other software. However, there are three areas in which special attention is required. These are in the definition of the application domain, the design of the architecture, and the implementation of the components.

1.4.1 Domain Definition

The applications derived from a generic architecture must be similar enough so that they may reasonably share a common design and set of components. The definition of the domain of the architecture must ensure that this is the case.

There are no hard and fast rules that govern the definition of a domain, but rather several things that should be considered. The first is whether the applications themselves are sufficiently similar. The domain of MacApp is all applications that might run on an Apple Macintosh. However, it is unlikely that an architecture intended for a real-time embedded application such as a missile guidance system would be particularly useful in the development of applications which might run on a graphics oriented work station like the Macintosh. This is because of such influences as the amount of software required to support a graphics interface with the user on a work station and the tight coupling between the real-time executive and the rest of the application in a missile guidance system.

The hardware and software environment is another major consideration. It is easier to develop reusable components which have an interface with the hardware or the operating system when those elements are common across the domain of the applications. This applies as well to other software interfaces such as those with a data base management system or with a graphics software package.

There must be enough applications in the domain to justify the cost of the development of a generic architecture. The number will vary with degree of variation in the applications supported by the architecture. For very similar applications, only a few would be needed to provide a viable domain. In this case, changes for individual applications would be likely to affect only a small share of the components. For domains with greater differences in the applications, the components would need to be more adaptable and the development cost would rise. A larger number of applications would be needed to offset this greater development cost.

Last, but not least, are the organizational and administrative considerations. The domain of the architecture should be within the scope of a single organization or project office. That is the organization that must provide the required funds, monitor the development of the architecture, resolve issues, support its use on applications, and receive the benefits of this approach to software development.

1.4.2 Design

Once a domain has been defined, it must be analyzed to identify the common requirements of the applications covered by the domain. This is the purpose of a "domain analysis". For a generic architecture, it should cover not only the applications, but also the hardware and software environment. For software with a potentially long life cycle, it is important to understand the types of changes in hardware, software, technology, and applications that it might have to accommodate in the future.

In discussing the decomposition of a system into modules, Parnas [7] proposes that the designer begin by listing difficult design decisions or design decisions which are likely to change. "Each module is then designed to hide such a decision from others." This is a particularly appropriate guideline for the design of the components of a generic architecture. With a generic architecture, change can be anticipated in the applications that use the architecture and in the hardware and software environment in which the applications execute. Within the software environment, independent changes can take place in the operating system, data base management system, graphics package, communications subsystem, and in any other area where there is an important software interface to the surrounding environment. The individual components should be designed to minimize the impact of changes in any of these areas, i.e., so that the minimum number of components are affected by any specific change.

It is highly unlikely that anyone has enough experience and ability to design a complete generic architecture without some prior experience with the domain covered by that architecture. Experience has shown that it is hard enough to produce a com-

[7] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol, 5, No. 12, p. 1053, December 1972.

plete design for a single complex application. A generic architecture presents many additional requirements, particularly if adequate attention is to be given to isolating the types of dependencies discussed above.

Two approaches to the development of a design are worth consideration. The first is to develop the architecture on the basis of the actual design of applications that have already been implemented. It is not unusual to find existing applications that are somewhat representative of those in the larger domain. This does not mean that the design should be the same. It is important to consider the other factors that have been discussed in this section. However, a great deal can be learned from a study of the requirements, interfaces, and algorithms of an existing design. In effect, the existing applications become early prototypes for the generic architecture.

The other approach is to develop a prototype of the architecture itself. Such an approach would attempt to develop early versions of all of the principal components, but would omit many of the minor ones. It could then be used to support the development of two or three actual applications. The results would be analyzed to identify changes to be made in the production version of the architecture as well as additional components to be made reusable.

Finally, the principles of object-oriented design should be used to define the objects around which each of the reusable components will be designed. There are likely to be objects for such things as the application itself (the top level of the design), windows, menus, entities retrieved from the data base, communications messages, etc. Each of these will be described by a set of variables and a set of operations that can be performed on the instances of each object. The packages that represent these objects will be the reusable components of the architecture.

1.4.3 Component Implementation

Two guidelines apply to the development of reusable components for a generic architecture. The first is to observe the generally accepted principles of object-oriented development in Ada. These principles will not be discussed further here as there are a number of good sources of information on this technique. References to some of them may be found in Appendix A.

The second guideline is to anticipate the adaptation requirements of the actual applications. The techniques of Section 1.3 should be used to facilitate the reuse of the components by anticipating the types of changes that may take place. Confine processing in less predictable areas to components that can be replaced where necessary.

Similar principles apply to the development of the components that are required to use a generic architecture to develop a specific application. If such a component appears in several different applications, it should probably be developed as a reusable component for use with the generic architecture itself.

1.5 Summary

The principal objectives of this chapter have been to define and analyze the concept of a generic architecture and to explore the use of Ada as a language for the implementation of its reusable components. The approach contained in a generic architecture has been used successfully in the past, particularly in the development of operating systems, report generators, and interactive applications for individual work stations. It is most appropriate where there are families of related applications with a similar information flow and a common hardware and software environment. Ada has been used successfully in the development of reusable components and has a number of features that would be useful in developing a set of integrated components that support a more complete design. It also has some known limitations of undetermined importance. Left unanswered are the following questions:

- How serious are the limitations of Ada, particularly with respect to inheritance?
- How well can dependencies within a generic architecture be isolated to maximize reuse?

- What limitations exist on the types of applications that can be supported with a generic architecture?

Adequate answers to these questions require actual experience in the development of generic architectures in Ada.

2. ARMY COMMAND AND CONTROL REQUIREMENTS

2.1 Introduction

This analysis of the requirements of command and control systems is based primarily on available information on the tactical portion of the Army Command and Control System (ACCS). Although this excludes command and control requirements unique to the other services and even to other systems of the Army itself. ACCS constitutes a potential domain for the possible application of a generic architecture.

2.1.1 Purpose and Scope

ACCS ties together battlefield systems for the following five functional segments:

1. Air defense.
2. Combat service support.
3. Fire support.
4. Intelligence and electronic warfare. and
5. Maneuver control.

With minor exceptions, these systems are intended for use by combat organizations at the battalion. brigade. division. corps. and force level.

Communications within and among the functional segments is supported by three additional systems:

- The PLRS/JTIDS Hybrid for real-time data distribution,

- Mobile Subscriber Equipment (MSE) for switched systems, and
- The Single Channel Ground and Airborne Radio System (SINCGARS) for combat net radio.

The interaction of these systems is illustrated in Figure 9.

The five functional segments of ACCS as well as the three communications systems are in various stages of development. Once available, it is expected that they will be maintained and enhanced for use by our battlefield forces well into the 21st century. Numerous changes can be anticipated over the life cycle of ACCS, both in Army doctrine, with its impact on force structure, information flows, and processing responsibilities, and in the technology of computer hardware, software, and command and control.

2.1.2 Major Influences

There are several things that will have a major influence on the ongoing development of the ACCS program. The most important of these are support for distributed operations, the use of common hardware and software, and the evolutionary development of the program itself.

Battlefield operations are inherently distributed. A task force is made up of many organizations at different levels of command, each with its own command post and command and control activities. Each commander must have enough information on subordinate, superior, lateral, allied, and enemy organizations to participate effectively in force operations. Information on the status of each friendly and enemy organization must be maintained at each interested command post to ensure timely response to processing requests. This also provides the redundancy necessary to compensate for the battlefield loss of command and control workstations. In addition, the command and control activities of higher level command posts are often geographically distributed to avoid a communications "signature" that could be used by the

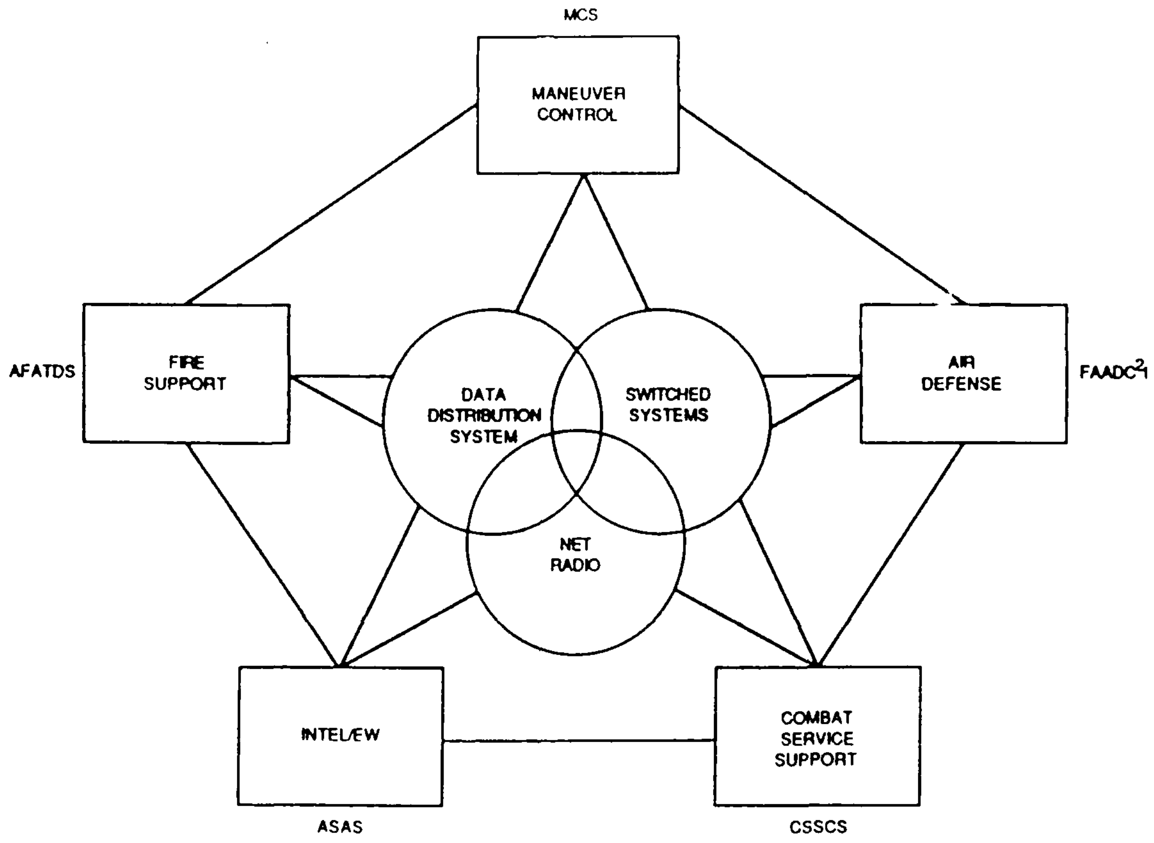

FIGURE 9. ACCS functional segments and communications support

enemy to identify key command centers.

The ACCS program is currently engaged in a major effort to acquire commercial, off-the-shelf, hardware and systems software for use by all five functional segments. This is a major change from past efforts which developed unique military sponsored equipment for each functional segment. The latter approach resulted in development efforts so lengthy that the systems were often obsolete before they were delivered, so costly that they could not be furnished to all of the required combat organizations, and so unique that interoperability became a serious problem.

Under the current acquisition effort, common workstations will be provided in three sizes, with varying degrees of military hardening. Except for the smallest, a hand held terminal, the stations will use a common operating system, data base management system, and graphics package. With minor exceptions, Ada will be used as the programming language in the development of all command and control software.

The development of ACCS will be evolutionary. This will be necessary to keep pace with changing hardware and software technology and to allow for necessary experimentation in the development of the technology of command and control. There is great potential in the use of artificial intelligence, powerful graphics, and other aids to the battlefield decision process. However, there is also a great deal of uncertainty as to what will really work under battlefield conditions. Techniques need to be incrementally developed, tested in battlefield exercises, subjected to the evaluation of experienced commanders, and revised at each step as command and control technology evolves.

This does not imply that command and control systems will be made available only at the end of a protracted evolutionary development effort. Command and control support for battlefield forces is needed now. The system must be released in segments that represent meaningful baselines in the evolution of ACCS.

2.2 Functional Requirements

The information in this section on the functional requirements of command and control systems is abstracted from the System Specification for the Maneuver Control System[8]. MCS is one of the five functional segments that make up ACCS. Although this analysis is based solely on the MCS system, it will be evident that the requirements apply, in some degree, to each of the other four functional segments. In addition, the MCS has a unique role among the five segments as the force integrator, i.e., it must integrate the activities of the individual segments to support the mission of the task force as a whole.

2.2.1 System Management

This function initializes a node (workstation) in a command and control network, manages the hardware resources of the node, and provides those services necessary to maintain continuous operations (CONOPS). Initialization includes the selective loading of the software, identification of the organizational unit associated with the node, and the entry of parameters required for the configuration and operation of the node. Resource management controls the use of peripheral devices, memory, the computer processor, and the scheduling mechanism. Continuous operations are maintained through facilities for graceful shutdown in the event of a power failure to preserve programs and data, reinitiation of operations, support for degraded operations when necessary, and support for operator initiated shutdown.

2.2.2 Data Base

This function includes services normally associated with a data base management system. It includes such things as the ability to:

[8]System Specification for Maneuver Constrol System, Type A, Ford Aerospace and Communications Corporation, June 1985

- Change the structure of a data base,
- Maintain a complete or partial data base at any node,
- Maintain independence from the programs that use the data,
- Maintain data at both local and remote locations,
- Transfer operations to alternate locations,
- Recover a data base, and
- Maintain consistency among data bases at different locations.

It maintains all of the data necessary to present the current situation and to support the requirements of the different battlefield functional segments. In the process. it must allow for the changing composition of both friendly and enemy forces In addition to data, it stores the information necessary to produce a wide assortment of standard messages.

A separate service data base is maintained to support the operation of the system itself. This contains such things as node and operator information, access control information, message queues, user files, and related material.

2.2.3 Information Processing

This major functional area includes the capabilities for the processing of messages, the generation and maintenance of data base information. and the generation of reports and displays based on information from the data base. Specific capabilities have been allocated to the following major processing categories.

Message Validation

This includes the processing necessary to ensure that messages received are valid and error free. Validity checking covers the message type. date and time, and destination.

Invalid messages are referred to the operator for review. It also acknowledges the receipt of messages to the originator.

Message Processing

This function manages the processing of both text and graphics messages within a node. It ensures that transmitted messages have been acknowledged, that messages are processed in the correct order, that multi-part messages are complete, that receiving nodes are in operation, that outgoing messages are properly queued for transmission, and that the operator is notified of the status of pending messages. It allows the operator to store information from a message into the data base and to print the contents of messages. Finally, it is responsible for updating information in active displays on the basis of incoming messages.

Message Routing

Information required for the routing and distribution of messages is maintained by this function. It keeps track of primary and alternate routes for each destination and informs the operator if a routing is not available.

Message Recording

A historical record is maintained of all message traffic. The operator has the ability to specify the retention period and recover messages from the history files.

Overload Processing

This function takes appropriate action when processing demands approach or exceed the capabilities of the system. This includes such things as the holding of messages on secondary storage devices, the suspension of low priority message processing, and notification to the operator of actions taken to reduce the overload.

Data Base Information Handling

The Data Base Information Handling function includes capabilities for the preparation of information for storage in the data base and for the preparation of requests for the retrieval of information from a data base. A data base may be updated either automatically or manually, on the basis of messages originating locally or at remote nodes. Retention periods may be specified for individual data items.

Statistical Analysis

This function provides the statistical tools needed to analyze data produced by other functions. Inferences can be drawn about system usage, system demand, error rates, communications channel usage and reliability, and electronic warfare activities.

2.2.4 Man-Machine Interface

The Man-Machine Interface includes those capabilities which allow the operator to interact with the system. It includes operator entry functions (keyboards, etc.) and operator display functions (displays, printers, etc.). The operator may:

- Enter commands through a keyboard or special function keys,
- Enter text and graphics through a keyboard, light pen, or other device,
- Display and make selections from menus, and
- Control the position of the cursor and the presentation of windows on the display screen.

In addition, this function supports the validation of data elements entered by the operator, alerts the operator to potential problems and errors, and controls prompts to the operator for input. The system must have the ability to display or print both graphics and text, or some combination of the two, using predefined formats.

2.2.5 Decision Support

This major area supports the commander and staff in the decision making process for combat operations. Pertinent data must be presented quickly and easily to assist in the analysis of current operations and support the production and distribution of plans and orders. Information is produced to aid the decision process by generating and disseminating current battlefield data via message traffic, standing requests for information and queries, and by analyzing current battlefield data. The capabilities necessary to support this function are organized into the following categories:

Query

A query is an operator initiated data search and retrieval from the data base of a local or remote node. It may require data from more than one node and be based on a variety of selection criteria. Response to a query may require the computation of totals, subtotals, percentages, and other numeric values. The operator may request an estimate of the size of the response to a query before it is processed. Selected data may be deleted from the data base of the local node.

Standing Request for Information

This is a query that is stored in a data base and initiated automatically in response to specified external conditions. They may be activated in response to incoming messages, updates to the data base, or timed events, such as the time of day, an elapsed time interval, or repeated interval.

Word Processing

This facility is needed for the creation and manipulation of preformatted messages, user-defined messages, plans, operation orders, and various other text files. It has the capabilities normally associated with a computer text editor.

Information Generation

This function includes the ability to create messages and reports with predefined formats, user-created formats, or manual input. It has the unique ability to combine information from the data base, graphics, and predefined formats with the word processing and graphics capabilities of the system to create messages and reports. Data from the data base of the local or a remote node can be automatically inserted into predefined messages and reports. A directory of predefined message formats can be displayed for operator selection. It can process for transmission any message generated by the operator and can automatically address standard messages for which distribution has been predetermined.

Graphics

The Graphics function will meet two operationally separate purposes. The first is map graphics, which includes the ability to display maps and situation overlays, as well as the ability to support battlefield simulations and scenario exercises. The second is decision graphics: the ability to provide the commander and staff with bar charts, histograms, and pie charts based on data contained in messages or the data base. The operator is allowed to execute a wide variety of editing operations on graphics displays, and to create, save, recall and manipulate Military Grid Reference locations. Maps may be displayed in different scales and annotated with standard symbols.

Decision Implementation Support

This function assists the user in the preparation and distribution of plans, orders, and other formatted information items. Also included is the ability to create, maintain, and manipulate a planning file data base, identical in structure to the current situation data base for use in planning activities.

Situation Analysis

This function monitors data and messages concerning the current tactical situation and detects predetermined situations and events that require some action on the part of the commander or staff. It also provides facilities for locating operation plans in the planning file data base, analyzing the relative strengths of friendly and enemy units, assisting in identifying alternative courses of action, calculation of movement time, and the simulation of alternate courses of action.

2.2.6 Communications Processing

The Communications Processing function supports those actions necessary to initiate, maintain, perform and control the communications functions of a command and control node. These are limited to those operations concerned with the actual transmission and reception of messages. It maintains information on the other nodes with which it might communicate. It has the ability to support both voice data communications over wire, combat net radio, and tactical multichannel communications channels using any of several different communications protocols. It keeps track of errors encountered in the communications process and takes appropriate action, e.g., switching to an alternate channel or media. It also provides a number of other capabilities which allow the operator or the system to control the communications process.

2.3 Other Requirements

In addition to the functional requirements which are derived directly from the command and control mission, there are a number of other requirements that must be met to ensure the integrity and effectiveness of the system. The discussion that follows on these requirements is also based largely on the MCS System Specification.

2.3.1 Security

Security requirements are associated with a number of the functions described in the previous section. A command and control system must be able to detect electronic warfare activities that threaten the integrity of force operations. It must control the handling of classified material, through such things as data base access controls, security classification markings on printouts and displays, the suppression of classified material from displays, the validation of message routing and security parameters, and the management of classified transmissions. In addition, it must be able to purge material from its memory and storage files to prevent disclosure by means that by-pass the security features of the system.

2.3.2 Training

It is likely that a command and control system will be used substantially more often in support of training activities and exercises than in support of battlefield operations. Support for training must be an integral part of any command and control system. The system will be used to train operators, either at a single node or in communication with other nodes, it must support field combat exercises, and it must be able to collect data on system use necessary to evaluate the performance of operators, commanders, and staff in training operations. It must be able to simulate message traffic and provide data base support for training. All of this must be done without affecting the combat readiness of the system.

2.3.3 Performance

Comparatively few performance requirements are identified in the MCS System Specification. Time limits in the range of 30 seconds to several minutes are provided for several functions. Message traffic of up to 500 single page messages per hour must be supported. Implied is keyboard-display response that does not detract from the efficient operation of a workstation, response equivalent to that of a good personal computer.

It is notable that these requirements are orders of magnitude less stringent than those found in many real-time embedded applications, where response times are often specified in milliseconds or less.

2.3.4 Flexibility

Flexibility, as it is defined here, applies to the ability to adapt the system to new requirements in the field. Excluded, is anything that might require a change to the software itself. The system must be adaptable through easily modified control and configuration parameters. It should be designed so that changes are likely to be localized and so that it can be easily adapted to changes in force structure, information flows, and information products.

2.3.5 Reliability

The system should be designed so that software errors are anticipated and cannot result in the failure of the system itself. It should have the ability to detect and respond appropriately to input errors, particularly in the keyboard entry of data by the operator.

2.3.6 Interoperability

Interoperability requirements are suggested by the current effort to procure common hardware and software for use in the five functional segments of the ACCS program. This, by itself, will result in the ability to transfer such things as the hardware and maintenance facilities among the functional segments.

However, it also provides an environment in which interoperability can be extended to the applications themselves, so that operators trained for one segment may more easily master the details of another. This applies as well to the ability of one organization to assume the role of another when required by the battlefield situation. It should also be possible to carry out the command and control responsibilities of several different nodes at a single node when conditions warrant such steps.

2.3.7 Maintainability

The components of the system should be modular and well documented, including the characteristics and limits of all data values. The Ada language should be used to implement the components of the system wherever possible. A longer term goal is to redesign and code non-Ada components in Ada. Changes made in the field may include program and data file patches and changes, but should not include changes to the source code of the programs.

2.3.8 Portability

The "goal" is to be able to transport applications from one hardware/software environment to another without modification of the source code. There should be no target computer or development system dependencies in the code. Ability to move subsets of the functions to a new hardware/software environment is a desirable feature.

2.3.9 Efficiency

A system is expected to make optimal use of its hardware resources. In particular, care must be taken to avoid uncontrolled use of such things as memory and disk storage space. However, the maintainability of a system is not to be sacrificed for the sake of efficiency.

2.4 Reusability Analysis

This section analyzes the functional requirements presented in Section 2.2 to determine the likely impact on the reusability of the resulting software. The analysis is intended to give a "qualitative feeling" for the degree to which reusable software could be used to implement such a system.

2.4.1 Dimensions of Reusability

It is useful to identify the dependencies that might be inherent in a software component to gain a better understanding of their impact on reusability. The important dependencies within the ACCS program are those that are associated with:

- A functional segment.
- A specific application within a functional segment,
- The hardware environment, and
- The software environment.

The specific nature of the dependencies in each area is discussed below.

Functional Segment Dependencies

These dependencies are associated with the five functional segments of the ACCS program. Processing that is unique to a functional segment, but may be common to several applications within that segment, would fall into this category.

Application Dependencies

Regardless of its use of reusable components, virtually every application performs some processing that is unique to the application. The components which perform that processing will be unique and will be developed as part of the effort to implement the application. Although those components are application dependent, they can generally be reused in other hardware and software environments as long as they are implemented in a language that is as environment independent as Ada.

Hardware Dependencies

Software components with hardware dependencies may be expected to change somewhat in the event of a future change to a new hardware architecture. They can also be affected by more limited changes to peripheral devices and other equipment included in the current configuration.

Software Dependencies

Dependencies in this area are related to the software that defines the execution environment. e.g., the operating system, data base management system, and the graphics package. Changes to this software will affect the components with which they have an interface. However, much of this software has been implemented with industry standard interfaces on different hardware. Therefore, a change in hardware does not necessarily mean that components with software dependencies will be affected.

2.4.2 Analysis by Function

This section analyzes the degree to which the components that implement a processing function are likely to be independent of the types of dependencies described above. Components that are highly independent of a particular dependency are also highly reusable across the boundaries associated with that dependency. The reusability of the software components for a function, under the circumstances associated with a dependency, is rated as high, medium, or low in the analysis that follows. The meaning attached to each of these rating is as follows:

- **High (H)** - The components associated with the function should be reusable with no more than minor changes in parameters.
- **Medium (M)** - The components should be reusable, but may require a significant amount of adaptation.
- **Low (L)** - It is unlikely that the components associated with the function can be reused.

The reusability ratings for each of the functional requirements are shown in Table I. In each case, a separate rating has been assigned for each type of dependency. A number used in place of a rating refers to an explanatory note at the end of the table. The individual ratings have been assigned on the basis of the descriptions contained in Section 2.2.

It is clear from the table that there are few dependencies suggested by the MCS requirements that would keep the software from being reused in the other ACCS functional segments as well. However, there are a number of possible dependencies on the hardware and software environment that could be troublesome in the future. Problems in that area can be reduced by isolating those dependencies in components that can be replaced when adapting to such changes and by using industry standard interfaces to the hardware and software environment wherever possible.

TABLE I. **Likelihood of Component Reuse Across ACCS Functions**

	Likelihood of Reuse Across			
	Segment	Application	Hardware	Software
System Management	H	H	L	L
Data Base	H	H	H	1
Information Processing				
Message Validation	H	H	H	H
Message Processing	H	H	H	M
Message Routing	H	H	H	M
Message Recording	H	H	H	M
Overload Processing	H	H	M	M
Data Base Information Handling	H	H	H	L
Statistical Analysis	H	H	H	H
Man-Machine Interface				
Operator Entry	H	H	2	3
Operator Display	H	H	4	5
Decision Support				
Query	H	H	H	6
Standing Request for Info.	H	H	H	L
Word Processing	H	H	H	7
Information Generation	H	H	H	H
Graphics	H	H	4	5
Decision Implementation Support	H	H	H	H
Situation Analysis	L	L	H	H
Communications Processing	H	H	L	8

Notes:

1. Reuse is unlikely if a change is made to a DBMS with a different program interface.

2. Would be affected by the introduction of new types of input devices.

3. Confirmation of input actions to the user could be affected by the introduction of a different graphics package.

4. Could be affected by the introduction of larger and higher resolution display devices.

5. Highly sensitive to a change in the graphics interface.

6. Highly sensitive to a change in the DBMS query interface.

7. The use of a text editor furnished with an operating system could be a problem if the operating system is replaced.

8. Could be sensitive to a change in the operating system.

2.4.3 Data Driven Operations

It should be noted that the functional requirements address general capabilities that are required in a command and control system rather than specific application details. They say nothing about such things as the content of specific messages or the details of individual decision aids. Instead they describe general mechanisms required to produce a wide assortment of messages or support different decision aids. This approach is necessary to reduce the complexity and enhance the stability of the software used to support command and control operations. It also tends to make the software components substantially more reusable.

The reason for this highly generalized approach can be found in the ACCS functional description[9]. That document identifies all of the different Army command and control units that are included in the higher level organizations of a force structure. It also identifies the command and control activities assigned to each type of unit. For each activity there is an information product, such as a plan or message. Each information product may be based on other information products, require coordination with other information products, and provide information required to develop other information products. Each information product is to be sent to a specified list of recipients. Thus, the information flows within a command and control network are highly dependent on the composition and organization of the task force supported by the network. They are also dependent on the types of information products prescribed by current Army doctrine as well as the particular requirements of the immediate tactical situation.

The software for ACCS must be able to produce, distribute, and analyze any type of message, report, or other information product required to support the operations of a task force. The very large number of these information products, combined with changes in force structure, organization, and information products, demands a "data driven" approach to the software support of command and control activities. That data driven approach is implicit in the functional requirements discussed above.

[9]The Command, Control, and Subordinate Systems (CCS2) Functional Description, The MITRE Corporation, October 1985

In this case, the distinguishing characteristic of a data driven approach is that the command and control software is not programmed to produce specific information products. Those are produced through the interaction of the operator, control information and data from the data base, and the processing software of the command and control system. The details of the individual information products are contained in text strings and other control information that is stored in the data base. The text strings provide templates or insertions for the information products. This allows information products to be added or changed as needed without changing the software. The changes are made only to the control information, templates, and insertions that are stored in the data base.

This data driven approach provides both the flexibility needed to meet changing command and control requirements and the generality required to use the software in each of the five functional segments of the ACCS program.

2.5 Summary

The complexity of task force structure, organization, and information flows has led to a data driven approach to the functional requirements, at least for the MCS program. Because the functional requirements of the MCS program have been expressed in that manner, they are likely to apply, to a large degree, to the other functional segments as well. However, it is also likely that the other segments will have additional unique requirements due to their more specialized missions. The other requirements discussed in Section 2.3 of this chapter - security, training, performance, etc. - are probably even more likely to apply to all of the ACCS functional segments. Where possible, software components developed to support the ACCS program should be designed with sufficient generality to meet the requirements of the entire program.

Changes in the hardware and software environment are a greater threat to the reusability of the software over the long run. Use of industry standard interfaces and the isolation of hardware and software dependencies in replaceable components is necessary to protect the Army's investment in ACCS software.

3. A STRAWMAN GENERIC ARCHITECTURE FOR COMMAND AND CONTROL

3.1 Purpose of the Strawman

Chapter 1 presented the concepts and characteristics of the generic architecture approach to reusable software. Chapter 2 presented the requirements of Army command and control systems, with particular emphasis on those that might be met through the use of reusable software. This chapter shows how a generic architecture might be designed to meet the requirements of Army command and control systems.

The "Strawman" generic architecture presented here is far from complete. A complete architecture is beyond the scope of this report. However, it is sufficient to illustrate the central concepts and characteristics of a generic architecture in the context of Army command and control requirements. The particular points from Chapter 1 that will be highlighted are the following:

1. The reusable components provided by a generic architecture represent abstract objects.
2. A component encapsulates both data and the operations on the data of the abstract object.
3. The components are highly integrated and intended to be used together.
4. Components that are oriented to an application domain and design can be larger and more complex.
5. Components must often be adapted to the specific requirements of an application.
6. Additional components, not provided with the architecture, are required for most applications.

Most of these points might apply to reusable software components in general and not to just those of a generic architecture. However, points 3 and 4, which deal with the interdependencies among the components and their size and complexity, are more uniquely associated with generic architectures. These are the traits that allow higher levels of reuse to be achieved through the use of generic architectures than with traditional libraries of reusable components.

The chapter begins with a description of the ACCS environment provided by the use of commercial, off-the-shelf, hardware and software across the five battlefield systems. It goes on to discuss the design and the components of the Strawman architecture. It then analyses the Strawman to show examples of the application of the concepts and characteristics identified here.

3.2 The ACCS Environment

The Army is currently in the process of procuring commercial hardware and software for use across the five battlefield segments of ACCS. This creates a common environment which will certainly increase the potential for the reuse of software components in the applications across the five segments.

3.2.1 Hardware

The hardware portion of the procurement calls for three types of systems: a handheld terminal unit, a portable computer unit that can be carried by one man, and a transportable computer unit which might be vehicle mounted or carried by a team of men. Only the portable and transportable systems must have the same instruction set architecture and share the same commercial software environment. The discussion here is confined to those two systems.

The following hardware components may be found on the common hardware:

- Processors, 1-3 MIPS,
- Floppy and hard disks,
- Archive device for the backup and recovery of files,
- Communications interface to a local area network and a variety of military devices and systems,
- Displays, color monitor, with both graphic and character display capabilities.
- Conventional alphanumeric keyboard with function keys, cursor control,
- Audible and visual alert capabilities,
- Programmable real time clock,
- Purgeable memory and disks,
- 1 to 16 megabytes of memory, and
- Printer with character and graphic capabilities.

3.2.2 Software

The software portion of the procurement includes a number of systems and packages. Several of these, e.g., a spread sheet and a word processing program, are required only for independent use on the portable computer unit and will not be used to support other command and control software. The software listed below provides the software environment for the execution of command and control applications:

- UNIX System V with real time extensions,
- A data base management system (DBMS) that includes support for SQL (Structured Query Language) and an Ada interface,
- Graphics software kernel (e.g., GKS), and
- Local area network and other communications software.

3.2.3 Man-Machine Interface

The procurement documents say relatively little about the man-machine interface. However, there is a clearly implied requirement for windows, menus, and other features that are comparable with those of a good commercial system. Text, charts, map graphics, and overlays must all be supported, both in black and white and in color. Highly sensitive cursor control, such as that provided by a mouse or light pen, is required to provide the precision in cursor movement needed to deal with high resolution graphics.

3.3 The Design

The design provided by the Strawman must be able to support a substantial amount of parallel processing. Operator activities required for such things as the preparation of outgoing messages must be handled simultaneously with the processing of incoming messages, the response to standing requests for information, and other activities that are not driven by operator commands. Each of these processes is implemented in the Strawman as a separate Ada task. Asynchronous tasks are also used in communications and other input/output operations to avoid processing delays and optimize the use of the computer processor.

Otherwise, the design is quite similar to that of personal computer applications that have a strong graphics oriented man-machine interface. Except for high priority alert or warning messages to the operator, the interaction between the man and the machine is handled as a single Ada task. Within that operator task, all activities are event driven. The display screen is used to provide feedback to the operator on such things as the entry of text or data, menu selections, and the movement of the cursor.

Further discussion of the Strawman design is limited to the operator task.

3.3.1 Flow of Control

The main event loop for the operator task is shown in Figure 10. The event manager is polled for the occurrence of an event that must be handled by the operator task. Events can include such things as the selection of an operator command from a menu, movement in the cursor control device, keyboard actions, or processing by other parts of the software which creates an event. Each polled event is analyzed to determine the required processing. The processing is performed and the event manager is polled for the next event.

Before the event loop is entered, the system goes through an initialization sequence during which the operator is logged on, files are opened, windows and menus are set up, and other tasks are initiated. The operator task then processes information in response to operator actions until an operator command event occurs that tells it to suspend processing and logoff the user.

A wide range of processing activities can occur in response to an event. If the event is the entry of an operator command through a menu selection or function key, then the processing will initiate the activities required by the command. If the event was caused by movement in the cursor control device, then the cursor position on the screen will be adjusted to reflect that movement. If keyboard or cursor action results in a window processing event, then the window will be displayed, scrolled, resized, or moved in the specified manner.

Figure 11 shows the processing of command events in greater detail. A routine that analyzes command events determines the type of processing that is required and invokes the necessary operations. Most of these operations are provided by the reusable components of the system.

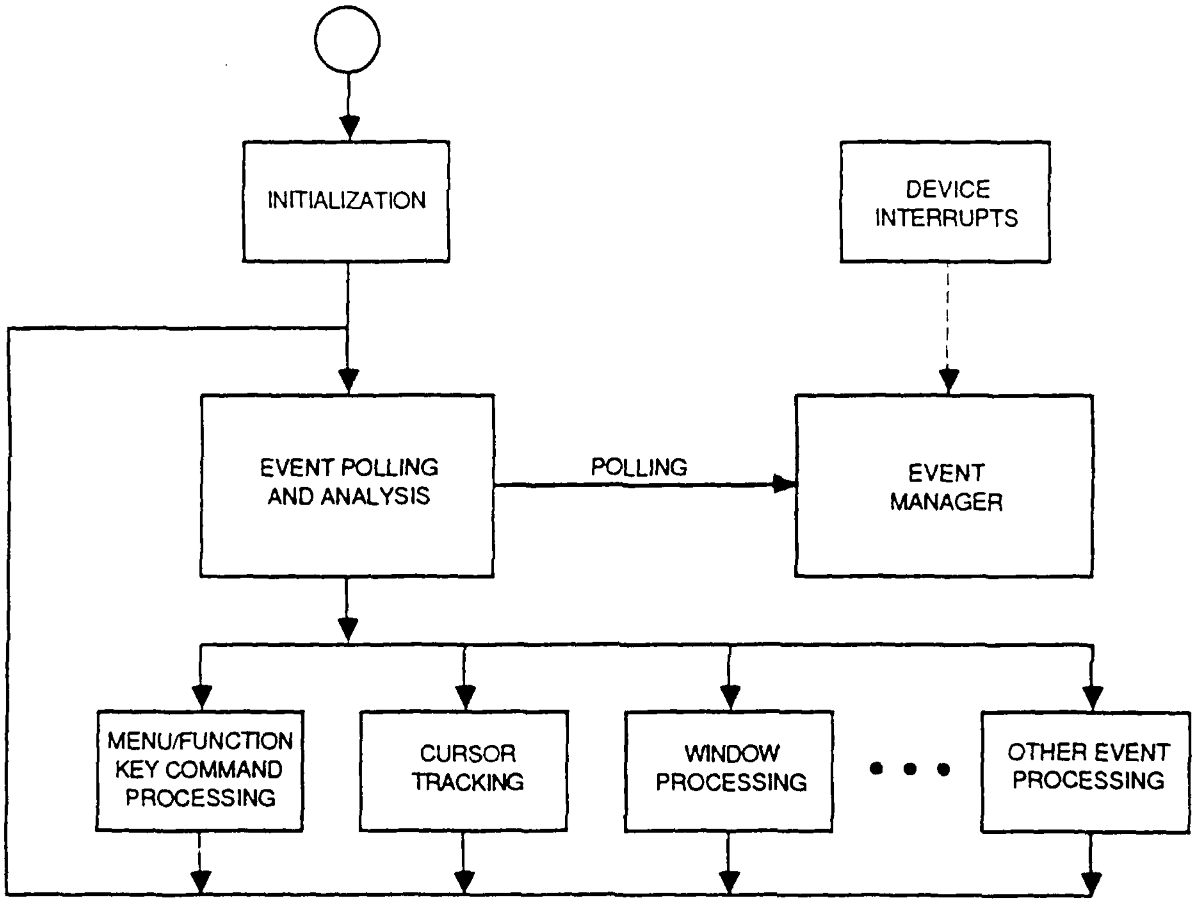

FIGURE 10. Main event loop for the operator task

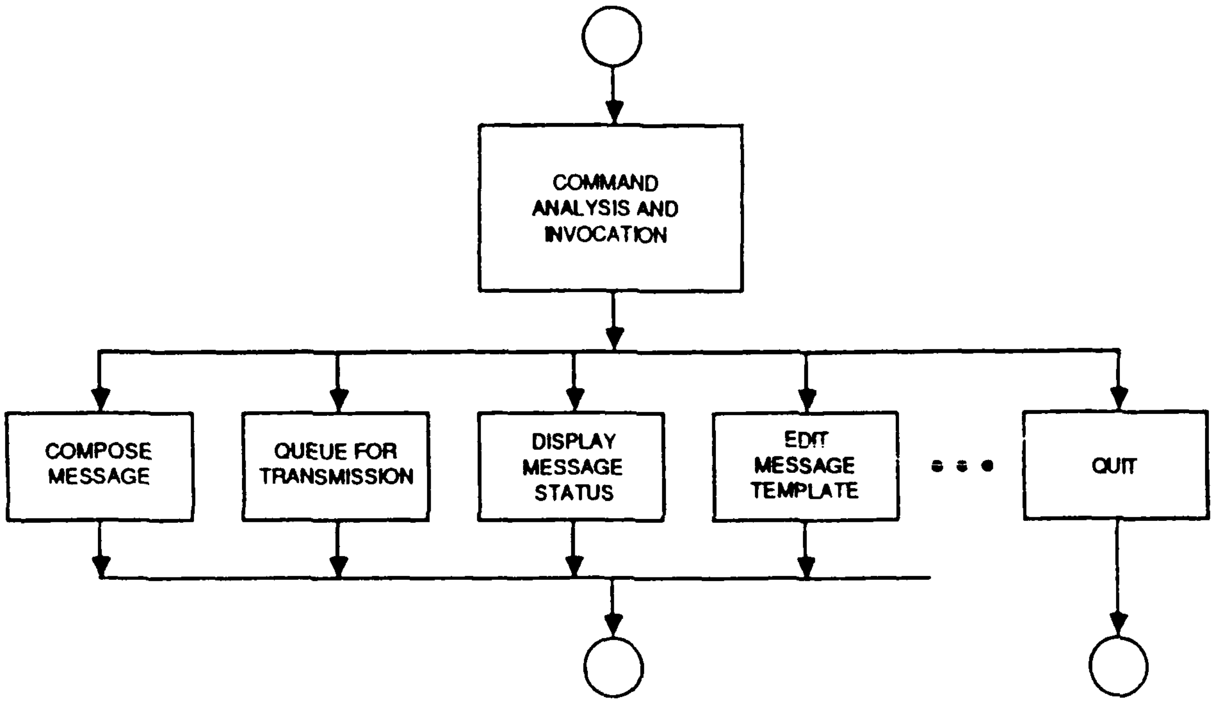

FIGURE 11. Menu/Function key command processing

3.3.2 Objects and Components

Each reusable component in based on an abstract object. An understanding of the role of each object is necessary to define the operations of the component as well as the data representation of the object necessary to support those operations. This section describes the objects represented by the components of the Strawman architecture, the data elements that might represent the object, and the types of operations that might be performed on the objects.

At this point, it is necessary to restrict the discussion of the architecture to a manageable number of objects and components. This will be done by selecting one major function, the preparation of outgoing messages by the operator, to illustrate the reusable component structure of the architecture. The objects, components, operations, and data representations discussed below are those that play a significant role in the preparation of outgoing messages. On that basis, the principal objects of interest are the following:

- Session,
- Data Base,
- Window,
- View,
- Dialog,
- Message,
- Text-String, and
- Channel.

In the Strawman, the reusable component that represents an object is identified as the manager of that object.

Session

The operations of a command and control system are carried out during a session, a period that is bounded by the logging on and the logging off of an operator. Data associated with a session includes the identification and security classification of the user, the node supported within the network, and the time that the operator logged on. The session manager records information about the session and makes that information available to other components on request.

Data Base

An instance of a data base object exists for each of the data bases in use on a node of a system. Separate data bases may exist for combat operations, planning, and training. The data representation of a data base object includes items, such as the data base type, which apply to the data base as a whole. It does not include the large amount of data that might be stored in a data base. The data base manager uses the commercial data base management system to process requests to store or retrieve data from a data base.

Window

A window is used to display information to the operator within a rectangular frame on the screen of a display. An application may have any number of windows. Those windows may be active or inactive, visible or hidden, at any point in time. One window may completely or partially hide another if they are both active on the screen at the same time. The window manager may create, delete, display, move, resize, hide, and perform other operations on windows.

View

A *view contains text, maps, overlays, and other material* for display in a window. The view manager uses the graphic operations of the commercial graphics package to fill in the portion of a view that is visible through a window. Views may be scrolled in any direction to expose other portions of a view.

Dialog

A dialog is a two way exchange of information between the computer and the operator. Dialogs are used to display warnings or alerts to an operator which must be acknowledged before the processing of the operator task can continue. They may also display templates or forms which must be filled in by the operator. Except for the unstructured text of a document, most data entry activities take place through dialogs. A dialog object is represented by data on the material to be displayed, the type of response expected, and *the criteria for checking the validity of the response*. The dialog manager supports different types of dialog displays, accepts different types of responses with an appropriate acknowledgement, and edits input data for errors.

Message

A message is any document or text string that might be sent or received from a node in a command and control network. The data representation includes such things as the message type, security classification, priority, sender identification, a list of recipients, the message text, and the current status of the message. The message manager composes, classifies, displays, sends, receives, acknowledges, and performs a wide variety of other operations on messages handled by applications derived from the Strawman architecture. Specific operations involved in the processing of outgoing messages are discussed in greater detail in Section 3.4.

Text String

A text string is a string of characters that may be used to express such things as messages, message templates, and addresses. Operations on text strings, e.g., insertion of data into a string, concatenation of text strings, etc., are performed by the text string manager.

Channel

A channel is the vehicle through which messages are passed from one node to another. Separate channels exist for each communications link between nodes. The channel manager controls the use of the communications channels, maintains information on c'.annel status, and uses the channels to transmit and receive messages.

3.4 The Processing of Outgoing Messages

The composition and transmission of an outgoing message is triggered by an event which may be an operator command, an incoming message, or the event associated with a standing request for information. The principal component involved in the handling of messages is the message manager. However, the operations of other components are invoked by the message manager to support its processing. This section examines the principal requirements that must be met in the processing of outgoing messages, the operations of the message manager that are necessary to meet those requirements, and the support provided to the message manager component by the operations of other components of the Strawman.

3.4.1 Principal Requirements for Outgoing Messages

Chapter 2 contained a significant number of requirements that applied to the processing of messages. Those that apply specifically to the processing that must be performed before a message can be transmitted are listed below:

1. Initiate a message in response to an operator command, incoming message, or a standing request for information.
2. Generate a message from a standard message template, a user defined template, or from operator input.
3. Identify the sender of a message.
4. Identify the recipients of a message.
5. Insert data from a data base into a message.
6. Assign the appropriate security classification and markings.
7. Determine the operational status of the receiving nodes.
8. Determine the appropriate routing for each recipient.
9. Inform the operator if a routing to a recipient is not available.
10. Insert a message into a queue for transmission.
11. Inform the operator of the status of messages that have been queued for transmission.
12. Ensure acknowledgement of receipt.
13. Archive each message for later reference.

The discussion of the operations based on these requirements deals specifically with messages that are initiated in response to an operator command and that use a standard message template.

3.4.2 Operations on Outgoing Messages

The sequence of operations required to prepare an outgoing message for transmission is initiated by the entry of a "prepare message" command by the operator. This event is passed on to the event analysis routines of the main event loop for the operator task. Those routines invoke the operations of the message manager that compose the message, assign the appropriate security classification, and display the message for review by the operator. When approved, the operator enters a "send message" command which results in the message being queued for transmission and archived for future reference. The message now comes under the control of tasks other than the operator task. Those tasks remove the message from the queue when the required channel is available, transmit the message, and record the receipt of an acknowledgement from the recipient.

Operations of the Message Manager Component

The operations of the message manager that are required to prepare such a message for transmission are shown in Table II. Each of these operations is implemented as an Ada subprogram, i.e., a procedure or function, that is accessed through the interface provided by an Ada package specification.

Operations of other Components

The Ada code that implements a subprogram for one of the operations in Table II may need to invoke operations provided by other reusable components. The immediate use of these other operations by the operations of the message manager is shown in Table III. The table also identifies the reusable component that provides each of those operations.

TABLE II. **Operations performed on messages by the message manager component**

Operation	Purpose
Compose Message	Composes an outgoing message on the basis of information entered by the operator, stored in the data base, and provided by other components.
Assign Security Classification	Assigns a security classification to an outgoing message on the basis of operator input or the security classification of information taken from the data base.
Review Message	Displays the full text of the message for operator review and approval.
Queue Message for Transmission	Inserts the message into the outgoing message queue for transmission.
Select Message for Transmission	Selects the highest priority message from the outgoing message queue for immediate transmission.
Archive Message	Inserts the message into the message archive file for possible later retrieval.
Record Message Acknowledgement	Records the acknowledgement of a message by a specified recipient.
Display Message Status	Displays the status of all outgoing messages for operator review.

TABLE III. **Operations performed on other components that are invoked by operations of the message manager component**

Message Manager Operation Invoked Operation (Invoked Component)	Reason for Invocation
Compose Message:	
Get Operator Input (Dialog Mgr.)	To obtain the message type, security classification, additional recipients, and other operator entered information.
Get Template (Data Base Mgr.)	To obtain the text string template that will be used to compose the message.
Get Distribution (Data Base Mgr.)	To get the standard list of recipients for the current type of message.
Get Address (Data Base Mgr.)	To obtain a nodal address and routing for each of the recipients.
Retrieve Data (Data Base Mgr.)	To retrieve data that is to be inserted into the text of the message as specified by the message template.
Insert Text (Text Mgr.)	To insert the retrieved data into the text string of the message template.
Identify Sender (Session Mgr.)	To obtain the identification of the sender/operator that was entered at the beginning of the current session.
Review Message:	
Compose Text View (View Mgr.)	To format a view of the message text that can be displayed in a window.
Display Text (Window Mgr.)	To display the view of the message text in a window on the display screen.
Display Status:	
Compose Text View (View Mgr.)	To format the text of the message status report for display.
Display Text (Window Mgr.)	To display the view of the message status report in a window on the display screen.

3.4.3 Component Dependencies

The use of the operations of a reusable component by the operations of another reusable component creates a dependency between the components that is common in generic architectures. It reflects the dependence on the design supplied by the architecture that allows larger and more complex reusable components to be developed.

The dependencies of the message manager on other components involved in the preparation and transmission of messages is shown in Figure 12. The message manager itself is invoked as part of the processing of at least two separate tasks. It is invoked by the command analysis routines of the operator task and by the channel manager component in the processing performed by the message transmission task. The channel manager invokes the "select message for transmission" operation of the message manager to obtain the next message that is available for transmission over an available channel.

3.5 Application of the Concepts of Generic Architectures

The introduction to this chapter listed six central concepts and characteristics of generic architectures that would be illustrated through the Strawman example. The remainder of the chapter reviews the application of those points to the Strawman design and components.

3.5.1 Components Represent Abstract Objects

Abstract objects provide an excellent basis for establishing the scope and content of a reusable component. For the component designer, they help to answer questions about the types of data and operations that should be included in a reusable component. For the user, they provide the basis for an understanding of the role of the component within a larger program. Much of the discussion of the Strawman architecture dealt with the Message Manager component which included all of the

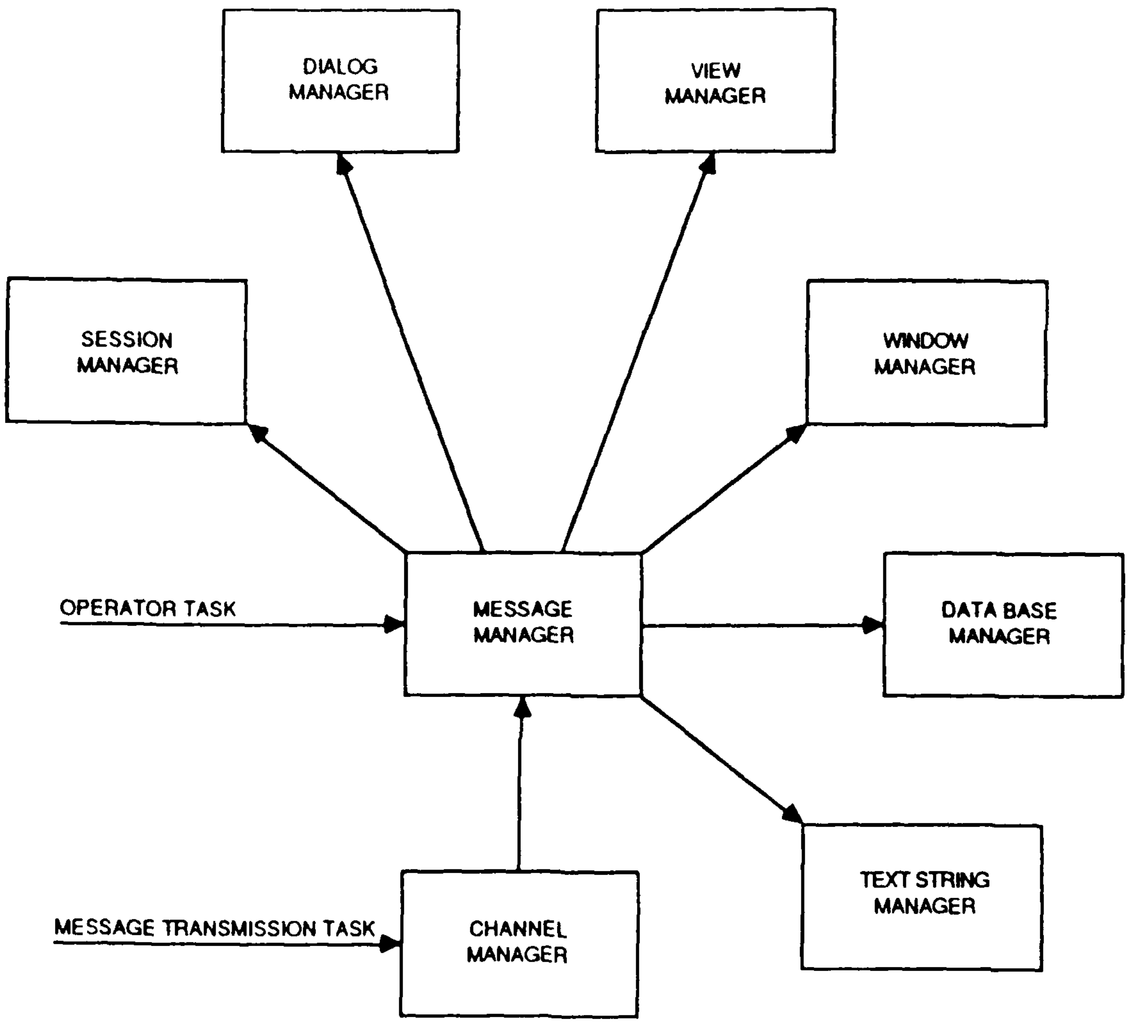

FIGURE 12. Component dependencies in the processing of outgoing messages

data and the operations that might be performed on an abstract representation of a *message*.

The representation of the message is abstract in the sense that the details of its internal data representation are of no importance to the user. The internal representation consists of a number of text strings and other data values that represent a real world message. The particular types of data that represent the message are sufficient to allow the software to carry out operations on such a representation in ways that parallel the operations that might be performed on a real world message outside of a computer system. Reuse of the message component in an application depends on whether that application must process messages in the manner supported by the component.

Most of the components identified in the discussion of the Strawman design were based on objects. However, the main event loop for the operator task does not have that object orientation. It is often useful to use the more process-oriented approach represented by the main event loop to tie together the reusable components of a system.

3.5.2 Encapsulation of Data and Operations

Encapsulation reinforces the concept of an abstract object. For example, the operations encapsulated in a reusable component for a message are those that must be able to manipulate the data used to represent a message. Conversely, those data values are hidden from operations that are not part of that reusable component. Thus, the user can then feel confident that all changes to the data used to represent a message will be made by operations that are included in the message manager component.

The practical effect of this is that the component is self-contained, a necessary condition for reusability. It can be used in different programs with the assurance that it will always behave in the same way, i.e., that there is nothing missing that might be required for the component to perform properly.

There is one important exception to this rule. The operations of a reusable component can invoke the operations of another component. This creates a dependency of one component on the other so that the using component cannot be used without a component that supplies the required operations. For example, the "compose message" operation of the message manager uses the "get operator input" operation of the dialog manager to obtain information needed to compose a message. To use the message manager component, one must also use a dialog manager component to supply the required operation.

Note, however, that it does not have to be the same dialog manager. The dialog manager component can be replaced with another dialog manager that supports an equivalent "get operator input" operation through the same interface. This will meet the needs of the message manager's "compose message" operation.

It is also important to note that the new "get operator input" operation does not have to perform the same processing as long as it produces the same result. If the keyboard display components of the computer hardware were replaced with a teletype, it would be possible to program a "get operator input" operation to carry out an equivalent operator dialog through the teletype. This would have no impact on the "compose message" operation of the message manager.

3.5.3 Integrated Nature of the Components

Dependencies among the reusable components of a generic architecture are common. The components are intended to be used together. Removal of a component requires that the component be replaced with another that will supply the required operations. The message manager component invokes directly a number of the operations of the other components. These, in turn, may invoke operations of yet other components, creating a network of dependencies among the components of a generic architecture.

The generic architecture clearly provides an integrated set of components that are intended to be used together in applications based on the architecture. The components are integrated in much the same way as the components of any application. The difference is in the emphasis that is placed on reusability. Each component plays a clearly defined role that is based on the object orientation of the component. Everything specific to the role of that component is encapsulated in that component. A component can be replaced or adapted without changing the components around it.

In simple terms, a generic architecture provides the user with an adaptable application. However, it may not be a complete application. The emphasis is on providing all of the components that are likely to be reused across the applications within the domain of the architecture. Components that will be different for each application are often represented by "dummy" versions which simply serve as place holders.

The dummy components can serve another purpose. They support the early testing of an application before all of the component adaptations and replacements have been made. Integration testing of the reusable components will have been done during the development of the architecture. Integration testing of a particular application may be done incrementally as the component adaptations and replacements are performed.

The final point that must be recognized is that the components are not intended to be reusable outside the application domain of the architecture. They are designed specifically to meet requirements that are common to most or all of the applications of the domain. It is this fact that permits the high degree of integration that one finds in the reusable components of a generic architecture.

3.5.4 Size and Complexity of the Components

Consider the number of operations identified in the Strawman example for the message object. These operations handle messages that require little more than simple text or data inserts. Purposely omitted are the operations for processing incoming messages as well as a number of initialization and other housekeeping operations. When the requirements of other types of messages are taken into account, the actual number of operations performed by a message manager reusable component might be an order of magnitude larger than that of the Strawman example.

The data representation of a message is also likely to be more complicated than shown in the Strawman. Among others, data items will be needed to:

- identify categories of messages that require different types of processing,
- support operations on incoming messages of different types,
- deal with the complexities of security classification and message priorities, and
- support the housekeeping operations of the message manager.

In addition, it is likely that the message manager would include a number of internal data items and structures used in message processing and for such things as the queues in which messages are held pending transmission or processing.

Thus, the reusable components of a generic architecture can be quite large and complex. They are complex in the different types of operations and the data that they encapsulate, and in their use of the operations of other components.

3.5.5 Adaptation of the Components

The ability to adapt a component to the specific requirements of an application is a key factor in its reusability. Without it, replacement of components within the architecture would be much more common. Within the message manager, adaptation might be anticipated in areas such as:

- the standard headings on messages, which might be specialized for each of the ACCS segments and for different categories of messages within a segment, and
- the types of data base data retrieval requests that might be generated for individual messages

The principal mechanisms provided by the Ada language for the adaptation of components were described in Chapter 1. In the examples above, the code necessary to specialize a heading or a data retrieval request would probably be contained in an Ada subprogram that could be made part of the message manager component through generic instantiation or through a separate sv unit.

Unfortunately, the Ada language forces the designer to anticipate most of the requirements for the adaptation of a component. If a subprogram is to be inserted into a reusable component through generic instantiation, then the reusable component must be d -loped as a generic component. Furthermore, the subprogram must be identified as a generic parameter that is supplied at the time that the reusable component is instantiated for use in an application. If a subprogram is to be treated as a separate subunit of a reusable component, then it must be identified as a separate subunit at the time that the component is developed.

This is where more complete support for the object-oriented concept of inheritance would be useful. Inheritance would allow any operation of a reusable component to be replaced and new operations and data to be added. All of these things could be done without changing the code of the original reusable component and without anticipating the need to make any of those changes. The replacements and additions would be contained in a new package that would be used with the original component in the application that required those specific adaptations. The Ada language does not have the features needed to support this approach to the adaptation of a reusable component.

This limitation becomes important when it is necessary to add new capabilities to an existing component. This might be necessary with the ongoing evolution of command and control technology. For example, support for "real time" dialogs between commanders at different nodes could have a major impact on the message manager component that probably would not have been anticipated in the original implementation.

Of course, the component can be replaced or updated to include the required new capabilities. However, other applications may still be using the original version of the component. It then becomes necessary to either maintain two versions of the message manager component or to insert the new version into all of the existing applications that use it.

When compared to the alternatives, the situation is not as bad as this analysis might suggest. It is unlikely that a reusable component as complex as the message manager *could be built without the design and other components provided* by a generic architecture. Thus, the traditional application-independent library approach to reusable components would probably not have been able to treat the message manager as a reusable component. It would then have been necessary to write application specific code for the handling of messages in each application. Reusable library components could still be used for some of the more primitive operations in this context, but the development and maintenance effort required would probably be significantly greater than that required to maintain two similar versions of the message manager.

3.5.6 Requirements for Additional Components

An application is likely to require additional components that are not provided with the generic architecture. For example, assume that a new class of messages is defined which includes one or more maps. The Strawman message manager does not draw maps and probably should not be extended to do operations on maps. Not only are map operations outside the average user's concept of the role of the message manager, but they are important in other contexts that have nothing to do with messages. The presentation of a map in a view for a situation display is one example.

A new map manager component might be the appropriate solution. One of the operations of the map manager would be to provide a digitized version of a map as a string of bytes, similar to a character string, that could be inserted into a message in much the same way as other text and data. Generic replacement of the "insert text" operation of the message manager would allow a new "insert text or map" operation to invoke the necessary map drawing operations of the map manager. It would provide the digitized representation of the map as a byte string that could be included in the outgoing message.

This is just an example of the way in which additional components might be added to meet the requirements of new applications. In the case of the map manager, it is likely that the component would be useful enough to be included in the generic architecture.

3.6 Summary

The objective of this chapter has been to illustrate the application of the concepts of a generic architecture through the use of examples based on the Strawman. That architecture addressed the specific requirements of Army command and control systems that were presented in Chapter 2.

Particular emphasis was placed on highlighting those characteristics of generic architectures that distinguish it from libraries of reusable components as an approach to software reuse. The components of a generic architecture are likely to be different in two important respects:

1. They are likely to be larger and more complex than the components of a library. and
2. They have dependencies on each other that lead to their use as an integrated set of components in applications supported by the architecture.

Both of these traits are a result of their orientation to the design and the application domain supported by the architecture. They are the traits that allow higher levels of software reuse to be achieved with generic architectures than with libraries.

Appendix A
BIBLIOGRAPHY

1. E. V. Berard, "An Object-Oriented Handbook for Ada Software," E.V.B. Software Engineering, Inc., 1985.

2. G. Booch, "Object-Oriented Development," IEEE Transactions on Software Engineering, vol. SE-12, no. 2, p. 211, February 1986.

3. F. P. Brooks, Jr., "No Silver Bullet, Essence and Accidents of Software Engineering," Computer, April 1987, p. 10.

4. G. D. Buzzard and T. N. Mudge, "Object-Based Computing and the Ada Programming Language," Computer, March 1985, p. 11.

5. E. W. Dijkstra, "Structured Programming" Software Engineering Techniques. Report on a Conference Sponsored by the NATO Science Committee, p. 84. October 1969.

6. E. Horowitz and J. B. Munson, "An Expansive View of Reusable Software." IEEE Transactions on Software Engineering," vol. SE-10, no. 5, p. 477, September 1984.

7. T. C. Jones, "Reusability in Programming: A Survey of the State of the Art." IEEE Transactions on Software Engineering, vol. SE-10, no. 5, p. 488, September 1984.

8. R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Designs and Code," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, p. 498, September 1984.

9. M. Mac an Airchinnigh, "Reusable Generic Packages Design Guidelines Based on Structural Isomorphism," Proceedings of the Annual Conference on Ada Technology 1985, p. 132.

10. Matsumoto Y., "Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels." IEEE Transactions on Software Engineering, vol. SE-10, no. 5, p. 502, September 1984.

11. J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, p. 564, September 1984.

12. D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, vol. 5, no. 12, p. 1053, December 1972.

13. D. L. Parnas, "On the Design and Development of Program Families," IEEE Transactions on Software Engineering, vol. SE-2, no. 1, p. 1. March 1976.

14. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, vol. SE-5, no. 2, p. 128, March 1979.

15. T. Ruegsegger, "RAPID: Reusable Ada Packages for Information System Development," Technology Strategies '87 Proceedings, January 1987.

16. K. J. Schmucker, "Object-Oriented Programming for the Macintosh," Hayden Book Company, 1986.

17. E. Seidewitz and M. Stark, "General Object-Oriented Software Development," Goddard Space Flight Center. August 1986.

18. E. Seidewitz, "Object-Oriented Programming in Smalltalk and Ada," Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications, October 1987.

19. T. A. Standish, "An Essay on Software Reuse," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, p. 494, September 1984.

20. H. L. Stern, "Comparison of Window Systems," Byte, November 1987, p. 265.

21. S. S. Yau *and* J. J. Tsai, "*A Survey of Software Design Techniques*," IEEE Transactions on Software Engineering, vol. SE-12, no. 6, p. 713, June 1986.

Appendix B DEMONSTRATION AND VALIDATION

B.1 Introduction

The study report developed the concept of a generic architecture and discussed its application to Army command and control systems. It provides most of the material needed to meet the requirements of the concepts exploration phase of the DOD-STD-2167 system life cycle.

This appendix describes the work that would be required to demonstrate and validate the generic architecture approach, *essentially the second phase of the system* life cycle. For the present effort, that phase would have two major objectives:

1. The demonstration and validation of the generic architecture approach to software development.
2. The development of a prototype generic architecture and a set of applications that use that architecture on a specific project.

The first objective has its own special requirements, quite different from those of the second. Demonstration of the approach requires that the details of the methodology be revealed at each step in the process and that the process be regularly reviewed and evaluated to provide a basis for further refinement of the methodology itself. Documentation of the "lessons learned" is a requirement at each step.

However, the first objective cannot be achieved without achieving the second. Successful development of a set of applications for a real project is necessary to show that the methodology really works. The development of a prototype generic architecture and a set of applications that use the architecture is the vehicle that supports the demonstration and validation of the methodology.

B.1.1 Role of the Sponsor

A generic architecture is inherently project specific. Its orientation to a specific application domain is a recurring theme in this study. It follows that its development must be sponsored or supported by the organization responsible for the project. No one else is expected to benefit from its development. No one else can provide adequate access to information on the requirements of the applications or evaluate the usefulness of the results.

Thus, the selection and involvement of an appropriate sponsor is a prerequisite to the effort described in this appendix. The demonstration and validation of the generic architecture approach cannot be accomplished without the support of the organization responsible for the applications that will be supported by the architecture. Execution of the process without the active participation of such a sponsor would seriously compromise the usefulness of the experience in the future development of generic architectures for other projects.

B.1.2 Tasks

This appendix provides the basis for a work plan for the demonstration and validation phase. For each task, it describes the purpose, the activities to be carried out, the issues to be resolved, and the products that will be delivered. The principal tasks are:

1. The domain analysis.

2. The development of a prototype generic architecture, and

3. The development of prototype applications, using the prototype generic architecture.

The activities of the demonstration and validation phase should be carried out under the standards provided by DOD-STD-2167 and its subordinate standards and data item descriptions. That will make it possible to evaluate the usefulness of those standards in the development of generic architectures and their applications. It will also provide an example of the tailoring of the standards to the particular requirements of this technology.

This document does not attempt to describe the 2167 process itself. Instead, it highlights the special considerations in the development of a generic architecture in each phase of that process.

B.2 Domain Analysis

B.2.1 Purpose

A domain analysis is conducted to establish the scope of the application domain supported by a generic architecture. Within the demonstration and validation phase, the domain analysis includes the effort normally associated with requirements analysis and preliminary design. Because the applications within the scope of the domain must share a common design and components, the top level design is needed to determine which applications are within the scope of the domain.

Within the present effort, the domain analysis will provide a vehicle for the development of the methodology to be used in future domain analyses for generic architectures. It will allow the methodology to be demonstrated and support an analysis of the process as well as the results. Finally, it will provide the results necessary to support the development of a prototype generic architecture and a set of prototype applications.

B.2.2 Problem Domain vs. Architectural Domain

However, the domain covered by the analysis, i.e., the problem domain, and the domain of the architecture are not necessarily the same. The problem domain includes all applications of interest to the sponsor of the analysis. The architectural domain includes all applications that can share a common design and a significant number of common components. The latter conditions may not be met by all of the applications in the problem domain.

It is clear that a problem domain may include several architectural domains. A group of applications within the problem domain may have enough similarities within the group to meet the requirements of an architectural domain, but have enough differences from other groups to prevent the sharing of a single architecture among the groups. In this case a separate architectural domain would exist for each group of applications. However, it should be noted that not all architectural domains may be important enough to justify the development of a generic architecture.

The remainder of this appendix will discuss the development of a single generic architecture for use on a significant number of the applications within a problem domain. It should be understood that the same approach would apply to the development of more than one generic architecture for the problem domain if that were necessary.

B.2.3 The Domain Analysis Process

A domain analysis should include the following five major activities:

1. The definition of the problem domain,
2. The identification of the software requirements for the applications within the problem domain,
3. Analysis of those software requirements to determine the appropriate approach to software reuse,

4. Development of the requirements and top level design for the reusable components, and

5. a cost benefit analysis for the reusable components.

It should be noted that the third activity is the selection of the approach to software reuse. A decision might be made to use a generic architecture or a library of reusable components. The requirements might suggest some other approach that might fall somewhere between those two, or a mix of those approaches. The general strategy, discussed here, is domain specific, i.e., it is based on the specific requirements of the problem domain. The above activities are appropriate for any domain specific strategy, regardless of the reusable software approach that is selected. They are discussed in more detail in the sections that follow.

The results of the domain analysis should be documented for review by the sponsor and others interested in the program. The documents should include:

- The System/Segment Specification (SSS) for the applications in the problem domain.
- Software Requirements Specifications (SRS's) for each application in the problem domain.
- An SRS and a Software Top Level Design Document (STLDD) for the reusable components that will be shared by the applications.
- A cost benefit analysis, and
- An analysis of the lessons learned in the domain analysis.

B.2.3.1 Definition of the Problem Domain

The problem domain includes all of the applications that will be considered in the domain analysis. Ideally, it should include all of the applications of concern to the sponsor. However, a number of factors may reduce the size of the problem domain. For example, some applications may be too small or temporary to benefit from the reuse of software components.

The problem domain should be identified and described in a System/Segment Specification. Within that document, each of the applications should be identified as a separate Computer Software Configuration Item (CSCI). An additional, separate CSCI should be identified for the reusable components that will be shared by the applications.

B.2.3.2 Identification of Application Software Requirements

DOD-STD-2167 requires a separate Software Requirements Specification (SRS) for each CSCI. Within such a document, the requirements should be organized to meet two objectives:

1. To support the decision on the approach to be taken to software reuse, and
2. To identify common functions that might be supported by reusable components.

Both of these objectives can be met if the organization of the functional requirements within an SRS anticipates the object-oriented design of a possible generic architecture. Under this arrangement, each of the major functions identified in an SRS would be approximately equivalent to an object-oriented reusable component in a generic architecture. To the extent allowed by the underlying requirements, the description of those major functions should be as similar as possible across the CSCI's so as to highlight both the similarities and the differences that must be recognized in the design of reusable components for those functions. Care must be taken to express these requirements in a style that will allow the sponsor to confirm that they represent a complete, correct, and adequate restatement of the original requirements.

There are several advantages to an object-oriented organization of the software requirements. As stated earlier, object-oriented components tend to be inherently reusable, and conversely, large reusable components are likely to be object oriented. Thus, an object-oriented organization of the requirements is more likely to highlight opportunities for the reuse of software components. It will also make it easier to determine whether the applications have enough in common to be supported by a generic architecture. Finally, it will also facilitate the traceability of the resulting design to the requirements because they will have the same object orientation.

It is clear that an object-oriented organization of the requirements requires some understanding of the design that will result from the requirements. This can happen only if a significant amount of work is done on the top level design of the generic architecture before the organization of the requirements is completed.

B.2.3.3 Determination of the Approach to Software Reuse

Once identified, the requirements should be reviewed to determine which of the requirements are common to a large share of the applications in the problem domain. They should also be reviewed to determine which applications are likely to be able to share the initial top level design that has been defined for a generic architecture. The selection of the generic architecture approach to software reuse is appropriate if:

1. The top level design can be shared by an adequate number of applications, and
2. Those applications have a significant number of similar functions.

The number of applications that must share a design and the number of functions that must be similar depends on the degree to which the applications and functions are similar. As few as two applications would be an adequate basis for a generic architecture if there were only minor differences in the applications.

It may also be appropriate to develop a library of reusable components that are not associated with a generic architecture. A library would be justified if the applications contained a number of functions that were common to two or more applications and that were relatively independent of the design of those applications.

A generic architecture should be developed if it can be justified on the basis of the above criteria and an analysis of the costs and benefits. The use of a common design and components across a number of applications is likely to reduce life cycle costs more than other approaches to software reuse. However, it need not preclude the use of component libraries or other approaches where they can support applications and functions not supported by a generic architecture. A separate CSCI, with its own Software Requirements Specification, should be created for the reusable components of other approaches because they will not share the top level design that is associated with the reusable components of a generic architecture.

It should now be possible to identify the specific applications within the domain of the generic architecture and thereby establish the scope of the architectural domain. This does not mean that new applications might not be added to that domain, but rather that it should be possible to determine whether the applications identified in the Software Segment Specification should be be included or excluded. Such a determination is needed to define the present scope of the architectural domain and to support the cost benefit analysis needed to support a decision on the implementation of the architecture.

B.2.3.4 Development of the Reusable Component Requirements and the Top Level Design

Software Requirements Specification(s) can now be developed for the reusable components. Each reusable component described in that document should be treated as a separate major function within the functional specifications.

Design requirements contained in an SRS for reusable components may reflect considerations not immediately evident in the functional requirements. In the case of a generic architecture, the design requirements should identify dependencies that should be isolated to facilitate later changes to the architecture and its applications. They must identify the parts of the design that may have to be adapted to the requirements of individual applications and the types of changes that may be required.

The next step is to complete the Software Top Level Design Document (STLDD) for the CSCI that covers the reusable components of the generic architecture. That document will describe the architecture, specify control relationships, identify data that is global to the architecture or to a reusable component, and describe in greater detail the requirements of each of the Top Level Computer Software Components (TLCSC's). In this case, a TLCSC is a reusable component of the generic architecture. Ada package specifications should be provided for each of the components. Those specifications should include the data representation of the object represented by the component and the declarations of any subprograms that implement operations on instances of the object. The subfunction performed by each operation should be described.

The top level design should specify the adaptation requirements of each of the components and of each of the operations within the components. It is also useful to provide an assessment of the likelihood that a component or operation will require adaptation for use in an application.

In the development of the design, there is a need to resist the temptation to build in generality that is not required by the requirements of the architectural domain. The efficient use of this approach depends on the ability to reduce complexity by tailoring the architecture to the specific requirement of its domain. Adaptability should not be provided unless there is a potential need for it in the development of applications within that domain.

B.2.3.5 Cost Benefit Analysis

A generic architecture is an approach to the development of a set of applications with similar requirements. The reuse of the design and components provided by the architecture can result in a significant reduction in the life cycle cost of those applications and produce other benefits as well. However, the development of a generic architecture is more expensive than the development of an application of similar size and complexity. That is due to the additional analysis and design effort that is required to meet the common requirements of all of the applications within the domain of the architecture and to provide the degree of adaptability that is needed for individual applications and future changes. The cost of this additional effort must be incurred before the architecture can be used to support its first application.

An analysis of the costs and benefits of a generic architecture should precede a decision on the implementation of such an architecture. In some respects, this task is made easier because the problem is bounded by the limits of the domain of the architecture. It is possible to make reasonable estimates of the number of applications that would benefit and the contribution that would be made to each application.

For purpose of the analysis, the benefits are likely to be both quantitative and qualitative. Available costing models can be quite useful in estimating the impact of the use of a generic architecture on the life cycle cost of the applications within he architectural domain. It is less easy to assess the long run impact of improved interoperability or greater adaptability to change. Never-the-less, it is important to explicitly assess both types of benefits before proceeding with the implementation.

The cost analysis is not much more difficult than any attempt to estimate the cost of a software system. It is necessary to estimate the life cycle cost of each of the applications both with and without an available generic architecture. The estimate with a generic architecture would be based on the number of lines of code that would have to be added or changed and the complexity of the task of extending the existing design provided by the architecture. I' would also require an estimate of the development cost of the architecture itself, based on estimating formulas for software of similar size and complexity. Allowance would have to be made for the additional effort required to carry out the requirements analysis, design, and testing of the architecture.

The impact of the qualitative benefits can only be described, but this should be done explicitly for the benefit of the decision maker. It should be possible to identify such things as:

- Situations where interoperability will be improved through the use of common components to implement message standards and the man-machine interface.
- The isolation of hardware and software dependencies that will facilitate the use of the architecture in other environments.

There may be problems to be identified as well. The use of a generic architecture can lessen the accountability of those responsible for implementing the applications. At least some of their problems may be beyond their control, if they are not also responsible for the implementation of the architecture. Of course, that problem is not significantly different from problems they might have with other existing software such as a data base management system or a graphics package.

B.3 Development of the Prototype Generic Architecture

B.3.1 Purpose

A prototype typically supports the principal functions of the final product, but omits the code required for exceptions or other special situations that can account for a substantial share of the total development cost. It is a logical first step in the evolutionary development of a software system.

The principal problem in the development of a prototype generic architecture is that it may not do anything. A generic architecture is essentially an incomplete application that provides a large share of the code that is required by any application based on the architecture. However, the code that provides the essence of the "principal functions" of an application is usually missing. Only the supporting code is in place. A generic architecture can be tested only by adding some of the code that would normally be provided by the using applications. Full validation of a prototype architecture requires the development of prototype applications that use the architecture.

However, the development of the prototype architecture will support the development and demonstration of the methodology. It also provides a basis for the subsequent development of applications that can be used to validate the architecture.

Not all of the requirements for the generic architecture will be met by the prototype. The functional requirements produced in the domain analysis should be reviewed to identify those that will be supported. Ideally, those that are omitted should be of relatively local importance and have little impact on the overall design. This will facilitate the later evolution of the prototype to the full scale architecture.

B.3.2 The Development Process

Starting with the results of the domain analysis, the work required to complete the development of a prototype generic architecture corresponds to the following phases of the DOD-STD-2167 software development cycle:

- Detailed Design,
- Code and Unit Testing,
- CSC Integration and Testing, and
- CSCI Testing.

The material that follows discusses the special considerations in the development of a generic architecture that affect each of these phases.

It is likely that the top level design that is produced as part of the domain analysis will be significantly more complete than most designs at this point in the development process. This is because a more detailed top level design for the architecture is needed to determine whether the architecture is adequate for use on a given application. This can make a difference in the scope of the architectural domain.

The principal activities that remain for the detailed design phase are:

1. The development of Program Design Language (PDL) for the implementation of the operations within the package body of each component,

2. The specification of variables and other data representations that are internal to each of the operations, and

3. The design of the adaptation mechanisms within each component.

These refinements in the design should be expressed in Ada code and comments that will evolve into the actual program source code for each component.

Most of the operations of an object-oriented reusable component should be treated as units within the Software Detailed Design Document (SDDD) produced by the detailed design process. The description of each unit should show how it may be adapted to meet the requirements of specific applications. This material is ultimately intended for use by the developers of applications that use the architecture. Here, it can serve as a guide to the developers of the components and as a basis for the independent development of tests to be used later in the formal testing of the reusable components.

The code and unit test phase will complete the evolution of the Ada code for each component that began with the top level design. Special attention should be given to the planning of unit tests for the object-oriented components to reduce the need for scaffolding to represent components that are not yet available. Additional effort will also be required to develop the code necessary to test the adaptation mechanisms of the components.

CSC integration and testing covers informal testing on aggregates of reusable components. It should be supported by a build plan for the components that allows the architecture to be used in the development of primitive applications as soon as enough components have been completed to support those applications. In the final stages, it may use the applications developed for use in the CSCI testing phase.

CSCI testing is the formal testing of the generic architecture, and it should be performed by an independent test team. The tests themselves should be based on documentation contained in the STLDD and the SDDD on the role and adaptation of each component. Final CSCI testing will be performed with the prototype applications.

The deliverables for the prototype architecture are the following:

- The SDDD for the reusable components of the generic architecture,
- The Ada source code for the architecture, and
- Test plans, procedures, and reports.

The analysis of the lessons learned should be performed in conjunction with the development of the prototype applications.

B.4 Development of The Prototype Applications

B.4.1 Purpose

The primary reason for the development of prototype applications that use the generic architecture is to validate the generic architecture itself. The development of the applications will provide a test of the reusability and adaptability of the design and the components provided by the architecture. It also provides an opportunity to evaluate the documentation that is provided to the user on the adaptation process.

The development of the prototype applications can also make an important contribution to the understanding of the technology. The development of the prototype applications will:

1. Develop and demonstrate the methodology needed for the development of applications that use a generic architecture,

2. Provide an initial set of applications that can be analyzed to gain a better understanding of the contribution made by the generic architecture, and

3. Support an analysis of the lessons learned in the development of applications based on a generic architecture.

In combination with the development of the prototype architecture, this task provides an opportunity to examine a generic architecture that has been implemented in Ada and to examine the use of that architecture in the development of the applications for which it was intended. It will provide a more objective basis for the comparison of this approach with other approaches to reusability and software development.

B.4.2 The Development Process

Several prototype applications are needed to adequately validate a generic architecture. The development of only one might suggest that the architecture was designed specifically for that application and that it might not be reusable on all of the applications of the domain. The prototypes selected from within the application domain should be sufficiently different to test the limits of the architecture and clearly demonstrate its reusability over its domain.

The functional requirements for each prototype application should be a subset of those identified for the application in the original domain analysis. Common requirements should be limited to those that are supported by the prototype generic architecture. Application specific requirements should be limited to those that test the adaptability of the architecture or that test the ability of the design to support the application.

The use of a generic architecture should streamline most of the remaining phases in the software development cycle. Preliminary design is needed only for significant extensions to the top level design provided by the architecture, for new components, and for replacements to components provided by the architecture. Detailed design is required for those items plus any significant adaptations of the reusable components.

Coding will be confined to items covered by the detailed design and any remaining adaptations of the reusable components provided by the architecture.

The availability of a tested generic architecture will simplify unit and integration testing. The architecture will provide much of the scaffolding and most of the drivers needed to support the testing process. To a large extent, the structure of the generic architecture will dictate the build plan.

The impact of the architecture will be less significant in the formal CSCI testing of the prototype applications. The independent test team must still work from the original SRS's for the applications. The use of a generic architecture in the development process has no impact on this requirement for independent testing. However, the task may be reduced somewhat by the identification of the common requirements for the architecture in the original domain analysis.

The principal deliverables for this task are:

- The STLDD and SDDD for each application,
- The source code for each application,
- Test plans, procedures, and reports,
- Object code for the tested applications, and
- An analysis of the lessons learned in the reuse of a generic architecture in the development of a family of related applications.

B.5 Summary

The technology represented by the use of generic architectures is still experimental. Elements of the approach are in use by SofTech on other projects and it is likely that other development organizations have similar approaches to the reuse of software components across applications. However, any expertise in the use of this approach is in the experience of the current practitioners and cannot be easily transferred to other organizations.

A major objective of the current effort is to develop an understanding of this evolving technology and present it in a form that will encourage and support its use on a more widespread basis. The prototype architecture and applications that are proposed in this chapter are useful as a way to illustrate the approach and examine its characteristics. The most useful deliverables may be those that deal with the methodology rather than the prototypes. However, the successful development of the prototypes is necessary to provide a real development environment in which to test the methodology.

The Army Command and Control System was used for analysis and illustration in the current study. The analysis was quite limited, and based on available documents and a few contacts with the staff of the ACCS program office. This was adequate to support the exploration of the concept in this phase of the work.

However, the demonstration and validation phase will require the active participation of a sponsor with the responsibility for the applications in the problem domain. The participation of a project office in the development of a prototype generic architecture will create the conditions necessary for an adequate test of the methodology and lay the groundwork for the subsequent development of a full scale generic architecture for use on that project. The sponsor's participation is critical to the success of the effort and the selection of an appropriate sponsor is a prerequisite to further work on this technology.

There are other issues, not covered in the present study, that should be examined when more experience has been gained in the application of this approach. These include such things as:

- The number of significantly different designs that are of importance in the development of generic architectures,

- The degree to which the design and components of a generic architecture can be used to support the development of applications outside the original domain of the architecture, and
- The degree to which reusable components can be shared among generic architectures.